

Contents

C++ Standard Library Reference

C++ Standard Library Header Files

`<algorithm>`

`<algorithm>` functions

`<allocators>`

`<allocators>` functions

`<allocators>` operators

`allocator_base` Class

`allocator_chunklist` Class

`allocator_fixed_size` Class

`allocator_newdel` Class

`allocator_suballoc` Class

`allocator_unbounded` Class

`allocator_variable_size` Class

`cache_chunklist` Class

`cache_freelist` Class

`cache_suballoc` Class

`freelist` Class

`max_fixed_size` Class

`max_none` Class

`max_unbounded` Class

`max_variable_size` Class

`rts_alloc` Class

`sync_none` Class

`sync_per_container` Class

`sync_per_thread` Class

`sync_shared` Class

`<array>`

`<array>` functions

<array> operators

array Class (C++ Standard Library)

<atomic>

atomic Structure

atomic_flag Structure

<atomic> functions

<atomic> enums

<bitset>

<bitset> operators

bitset Class

<cassert>

<ccomplex>

<cctype>

<cerrno>

<cfenv>

<cfloat>

<chrono>

<chrono> functions

<chrono> operators

chrono literals

common_type Structure

duration Class

duration_values Structure

steady_clock struct

system_clock Structure

time_point Class

treat_as_floating_point Structure

<cinttypes>

<ciso646>

<climits>

<locale>

<cmath>

<codecvt>

<codecvt> enums

codecvt_utf8

codecvt_utf8_utf16

codecvt_utf16

<complex>

<complex> functions

<complex> operators

complex Class

complex<double>

complex<float>

complex<long double>

<condition_variable>

<condition_variable> enums

condition_variable Class

condition_variable_any Class

<csetjmp>

<csignal>

<cstdarg>

<cstdbool>

<cstddef>

<cstdint>

<cstdio>

<cstdlib>

<cstring>

<ctgmath>

<ctime>

<cvt-wbuffer>

<cvt-wstring>

<wchar>

<wctype>

<deque>

<deque> functions

<deque> operators

deque Class

<exception>

<exception> functions

<exception> typedefs

bad_exception Class

exception Class

<filesystem>

<filesystem> operators

<filesystem> functions

<filesystem> enumerations

directory_entry Class

directory_iterator Class

file_status Class

filesystem_error Class

path Class

recursive_directory_iterator Class

space_info Structure

<forward_list>

<forward_list> functions

<forward_list> operators

forward_list Class

<fstream>

<fstream> typedefs

basic_filebuf Class

basic_fstream Class

basic_ifstream Class

basic_ofstream Class

<functional>

<functional> functions

<functional> operators

_1 Object

bad_function_call Class

binary_function Struct

binary_negate Class

binder1st Class

binder2nd Class

const_mem_fun_ref_t Class

const_mem_fun_t Class

const_mem_fun1_ref_t Class

const_mem_fun1_t Class

divides Struct

equal_to Struct

function Class

greater Struct

greater_equal Struct

hash Class

is_bind_expression Class

is_placeholder Class

less Struct

less_equal Struct

logical_and Struct

logical_not Struct

logical_or Struct

mem_fun_ref_t Class

mem_fun_t Class

mem_fun1_ref_t Class

mem_fun1_t Class

minus Struct

modulus Struct

multiplies Struct

negate Struct

not_equal_to Struct

plus Struct

pointer_to_binary_function Class

pointer_to_unary_function Class

reference_wrapper Class

unary_function Struct

unary_negate Class

<future>

<future> functions

<future> enums

future Class

future_error Class

is_error_code_enum Structure

packaged_task Class

promise Class

shared_future Class

uses_allocator Structure

<hash_map>

<hash_map> functions

<hash_map> operators

hash_compare Class

hash_map Class

hash_multimap Class

value_compare Class

<hash_set>

hash_set Class

hash_multiset Class

<hash_set> functions

<hash_set> operators

<initializer_list>

initializer_list Class

<iomanip>

<iomanip> functions

<ios>

<ios> functions

<ios> typedefs

basic_ios Class

fpos Class

ios_base Class

<iosfwd>

<iostream>

<istream>

<istream> functions

<istream> operators

<istream> typedefs

basic_iostream Class

basic_istream Class

<iterator>

<iterator> functions

<iterator> operators

back_insert_iterator Class

bidirectional_iterator_tag Struct

checked_array_iterator Class

forward_iterator_tag Struct

front_insert_iterator Class

input_iterator_tag Struct

insert_iterator Class

istream_iterator Class

istreambuf_iterator Class

iterator Struct

iterator_traits Struct

move_iterator Class

ostream_iterator Class

ostreambuf_iterator Class

output_iterator_tag Struct

random_access_iterator_tag Struct

reverse_iterator Class

unchecked_array_iterator Class

<limits>

<limits> enums

numeric_limits Class

<list>

<list> operators

list Class

<locale>

<locale> functions

codecvt Class

codecvt_base Class

codecvt_byname Class

collate Class

collate_byname Class

ctype Class

ctype<char> Class

ctype_base Class

ctype_byname Class

locale Class

messages Class

messages_base Class

messages_byname Class

money_base Class

money_get Class

money_put Class

moneypunct Class

moneypunct_byname Class

num_get Class

num_put Class

numpunct Class

numpunct_byname Class

time_base Class

time_get Class

time_get_byname Class

time_put Class

time_put_byname Class

wbuffer_convert Class

wstring_convert Class

<map>

<map> functions

<map> operators

map Class

multimap Class

value_compare Class (<map>)

<memory>

<memory> functions

<memory> operators

<memory> enums

allocator Class

allocator<void> Class

allocator_traits Class

auto_ptr Class

bad_weak_ptr Class

enable_shared_from_this Class

pointer_traits Struct

raw_storage_iterator Class

shared_ptr Class

unique_ptr Class

weak_ptr Class

<mutex>

<mutex> functions

adopt_lock_t Structure

defer_lock_t Structure

lock_guard Class

mutex Class (C++ Standard Library)

once_flag Structure

recursive_mutex Class

recursive_timed_mutex Class

timed_mutex Class

try_to_lock_t Structure

unique_lock Class

<new>

<new> functions

<new> typedefs

<new> operators

bad_alloc Class

nothrow_t Structure

<numeric>

<numeric> functions

<ostream>

basic_ostream Class

<ostream> functions

<ostream> operators

<ostream> typedefs

<queue>

<queue> operators

priority_queue Class

queue Class

<random>

<random> functions

bernoulli_distribution Class

binomial_distribution Class

cauchy_distribution Class

chi_squared_distribution Class

discard_block_engine Class
discrete_distribution Class
exponential_distribution Class
extreme_value_distribution Class
fisher_f_distribution Class
gamma_distribution Class
geometric_distribution Class
independent_bits_engine Class
linear_congruential_engine Class
lognormal_distribution Class
mersenne_twister_engine Class
negative_binomial_distribution Class
normal_distribution Class
piecewise_constant_distribution Class
piecewise_linear_distribution Class
poisson_distribution Class
random_device Class
seed_seq Class
shuffle_order_engine Class
student_t_distribution Class
subtract_with_carry_engine Class
uniform_int_distribution Class
uniform_real_distribution Class
weibull_distribution Class

<ratio>

<regex>

<regex> functions

<regex> operators

<regex> typedefs

basic_regex Class

match_results Class

regex_constants namespace

- regex_error Class
- regex_iterator Class
- regex_token_iterator Class
- regex_traits Class
- regex_traits<char> Class
- regex_traits<wchar_t> Class
- sub_match Class
- <scoped_allocator>
 - <scoped_allocator> operators
 - scoped_allocator_adaptor Class
- <set>
 - <set> functions
 - <set> operators
 - set Class
 - multiset Class
- <shared_mutex>
- <sstream>
 - <sstream> functions
 - <sstream> typedefs
 - basic_stringbuf Class
 - basic_istringstream Class
 - basic_ostringstream Class
 - basic_stringstream Class
- <stack>
 - <stack> operators
 - stack Class
- <stdexcept>
 - domain_error Class
 - invalid_argument Class
 - length_error Class
 - logic_error Class
 - out_of_range Class

[overflow_error Class](#)

[range_error Class](#)

[runtime_error Class](#)

[underflow_error Class](#)

[<streambuf>](#)

[<streambuf> typedefs](#)

[basic_streambuf Class](#)

[<string>](#)

[<string> functions](#)

[<string> operators](#)

[<string> typedefs](#)

[basic_string Class](#)

[char_traits Struct](#)

[char_traits<char> Struct](#)

[char_traits<char16_t> Struct](#)

[char_traits<char32_t> Struct](#)

[char_traits<wchar_t> Struct](#)

[<string_view>](#)

[<string_view> operators](#)

[<string_view> typedefs](#)

[basic_string_view Class](#)

[hash<string_view> Specialization](#)

[<sstream>](#)

[sstrstreambuf Class](#)

[istrstream Class](#)

[ostrstream Class](#)

[strstream Class](#)

[<system_error>](#)

[<system_error> functions](#)

[<system_error> operators](#)

[<system_error> enums](#)

[error_category Class](#)

[error_code Class](#)

[error_condition Class](#)

[is_error_code_enum Class](#)

[is_error_condition_enum Class](#)

[system_error Class](#)

[<thread>](#)

[<thread> functions](#)

[<thread> operators](#)

[hash Structure \(C++ Standard Library\)](#)

[thread Class](#)

[<tuple>](#)

[<tuple> functions](#)

[<tuple> operators](#)

[tuple Class](#)

[tuple_element Class <tuple>](#)

[tuple_size Class <tuple>](#)

[<type_traits>](#)

[add_const Class](#)

[add_cv Class](#)

[add_lvalue_reference Class](#)

[add_rvalue_reference Class](#)

[add_pointer Class](#)

[add_volatile Class](#)

[aligned_storage Class](#)

[aligned_union Class](#)

[alignment_of Class](#)

[common_type Class](#)

[conditional Class](#)

[decay Class](#)

[enable_if Class](#)

[extent Class](#)

[integer_sequence Class](#)

integral_constant Class, bool_constant Class

invoke_result Class

is_abstract Class

is_arithmetic Class

is_array Class

is_assignable Class

is_base_of Class

is_class Class

is_compound Class

is_const Class

is_constructible Class

is_convertible Class

is_copy_assignable Class

is_copy_constructible Class

is_default_constructible Class

is_destructible Class

is_empty Class

is_enum Class

is_final Class

is_floating_point Class

is_function Class

is_fundamental Class

is_integral Class

is_invocable classes

is_literal_type Class

is_lvalue_reference Class

is_member_function_pointer Class

is_member_object_pointer Class

is_member_pointer Class

is_move_assignable Class

is_move_constructible Class

is_nothrow_assignable Class

is_nothrow_constructible Class
is_nothrow_copy_assignable Class
is_nothrow_copy_constructible Class
is_nothrow_default_constructible Class
is_nothrow_destructible Class
is_nothrow_move_assignable Class
is_nothrow_move_constructible Class
is_null_pointer Class
is_object Class
is_pod Class
is_pointer Class
is_polymorphic Class
is_reference Class
is_rvalue_reference Class
is_same Class
is_scalar Class
is_signed Class
is_standard_layout Class
is_trivial Class
is_trivially_assignable Class
is_trivially_constructible Class
is_trivially_copy_assignable Class
is_trivially_copy_constructible Class
is_trivially_copyable Class
is_trivially_default_constructible Class
is_trivially_destructible Class
is_trivially_move_assignable Class
is_trivially_move_constructible Class
is_union Class
is_unsigned Class
is_void Class
is_volatile Class

- make_signed Class
- make_unsigned Class
- rank Class
- remove_all_extents Class
- remove_const Class
- remove_cv Class
- remove_extent Class
- remove_pointer Class
- remove_reference Class
- remove_volatile Class
- result_of Class
- underlying_type Class
- <type_traits> functions
- <type_traits> typedefs
- <typeindex>
 - hash Structure
 - type_index Class
- <typeinfo>
- <unordered_map>
 - <unordered_map> functions
 - <unordered_map> operators
 - unordered_map Class
 - unordered_multimap Class
- <unordered_set>
 - <unordered_set> functions
 - <unordered_set> operators
 - unordered_set Class
 - unordered_multiset Class
- <utility>
 - <utility> functions
 - <utility> operators
 - identity Structure

pair Structure

<valarray>

<valarray> functions

<valarray> operators

gslice Class

gslice_array Class

indirect_array Class

mask_array Class

slice Class

slice_array Class

valarray Class

valarray<bool> Class

<vector>

vector Class

vector<bool> Class

vector<bool>::reference Class

vector<bool>::reference::flip

vector<bool>::reference::operator bool

vector<bool>::reference::operator=

<vector> functions

<vector> operators

C++ Standard Library Overview

Using C++ Library Headers

C++ Library Conventions

C++ Program Startup and Termination

Safe Libraries: C++ Standard Library

_ITERATOR_DEBUG_LEVEL

_SCL_SECURE_NO_WARNINGS

Checked Iterators

_SECURE_SCL

Debug Iterator Support

_HAS_ITERATOR_DEBUGGING

Thread Safety in the C++ Standard Library

stdext Namespace

C++ Standard Library Containers

<sample container>

<sample container> Operators

operator!=

operator== (<sample container>)

operator< (<sample container>)

operator<= (<sample container>)

operator> (<sample container>)

operator>=

<sample container> Specialized Template Functions

swap (<sample container>)

<sample container> Classes

Sample Container Class

Sample Container Members

Sample Container Typedefs

Sample Container Member Functions

Iterators

Algorithms

Allocators

Function Objects in the C++ Standard Library

iostream Programming

What a Stream Is

Input-Output Alternatives

Output Streams

Constructing Output Stream Objects

Using Insertion Operators and Controlling Format

Output File Stream Member Functions

Effects of Buffering

Binary Output Files

Overloading the << Operator for Your Own Classes

Writing Your Own Manipulators Without Arguments

Input Streams

Constructing Input Stream Objects

Using Extraction Operators

Testing for Extraction Errors

Input Stream Manipulators

Input Stream Member Functions

Overloading the >> Operator for Your Own Classes

Input-Output Streams

iostreams Conventions

Custom Manipulators with Arguments

Output Stream Manipulators with One Argument (int or long)

Other One-Argument Output Stream Manipulators

Regular Expressions (C++)

File System Navigation

C++ Standard Library Reference

3/11/2019 • 2 minutes to read • [Edit Online](#)

A C++ program can call on a large number of functions from this conforming implementation of the C++ Standard Library. These functions perform essential services such as input and output and provide efficient implementations of frequently used operations.

For more information about Visual C++ run-time libraries, see [CRT Library Features](#).

In This Section

[C++ Standard Library Overview](#)

Provides an overview of the Microsoft implementation of the C++ Standard Library.

[iostream Programming](#)

Provides an overview of iostream programming.

[Header Files Reference](#)

Provides links to reference topics discussing the C++ Standard Library header files, with code examples.

C++ Standard Library Header Files

5/7/2019 • 2 minutes to read • [Edit Online](#)

Header files for the C++ Standard Library and extensions, by category.

Headers by category

CATEGORY	HEADERS
Algorithms	<algorithm>
C library wrappers	<cassert> , <cctype> , <cerrno> , <cfenv> , <cmath> , <cinttypes> , <ciso646> , <climits> , <locale> , <cmath> , <csetjmp> , <csignal> , <stdarg> , <stdbool> , <stddef> , <stdint> , <stdio> , <stdlib> , <string> , <tgmath> , <time> , <wchar> , <wctype>
Containers	
Sequence containers	<array> , <deque> , <forward_list> , <list> , <vector>
Ordered associative containers	<map> , <set>
Unordered associative containers	<unordered_map> , <unordered_set>
Adaptor containers	<queue> , <stack>
Errors and exception handling	<exception> , <stdexcept> , <system_error>
I/O and formatting	<filesystem> , <fstream> , <iomanip> , <ios> , <iosfwd> , <iostream> , <istream> , <ostream> , <sstream> , <streambuf> , <stringstream>
Iterators	<iterator>
Localization	<codecvt> , <cv/wbuffer> , <cv/wstring> , <locale>
Math and numerics	<complex> , <limits> , <numeric> , <random> , <ratio> , <valarray>
Memory Management	<allocators> , <memory> , <new> , <scoped_allocator>
Multithreading	<atomic> , <condition_variable> , <future> , <mutex> , <shared_mutex> , <thread>
Other utilities	<bitset> , <chrono> , <functional> , <initializer_list> , <tuple> , <type_traits> , <typeinfo> , <typeid> , <utility>

CATEGORY	HEADERS
Strings and character data	<regex> , <string> , <string_view>

See also

[Using C++ Library Headers](#)

[C++ Standard Library](#)

<algorithm>

10/31/2018 • 12 minutes to read • [Edit Online](#)

Defines C++ Standard Library container template functions that perform algorithms.

Syntax

(see relevant links below for specific algorithm syntax)

Remarks

The C++ Standard Library algorithms are generic because they can operate on a variety of data structures. The data structures that they can operate on include not only the C++ Standard Library container classes such as `vector` and `list`, but also program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm. C++ Standard Library algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators.

C++ Standard Library algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all pointers in the ranges must be dereferenceable and, within the sequences of each range, the last position must be reachable from the first by incrementation.

The C++ Standard Library algorithms extend the actions supported by the operations and member functions of each C++ Standard Library container and allow working, for example, with different types of container objects at the same time. Two suffixes have been used to convey information about the purpose of the algorithms.

- The `_if` suffix indicates that the algorithm is used with function objects operating on the values of the elements rather than on the values of the elements themselves. The `find_if` algorithm looks for elements whose values satisfy the criterion specified by a function object, and the `find` algorithm searches for a particular value.
- The `_copy` suffix indicates that the algorithm not only manipulates the values of the elements but also copies the modified values into a destination range. The `reverse` algorithm reverses the order of the elements within a range, and the `reverse_copy` algorithm also copies the result into a destination range.

C++ Standard Library algorithms are often classified into groups that indicate something about their purpose or requirements. These include modifying algorithms that change the value of elements as compared with non-modifying algorithms that do not. Mutating algorithms change the order of elements, but not the values of their elements. Removing algorithms can eliminate elements from a range or a copy of a range. Sorting algorithms reorder the elements in a range in various ways and sorted range algorithms only act on ranges whose elements have been sorted in a particular way.

The C++ Standard Library numeric algorithms that are provided for numerical processing have their own header file `<numeric>`, and function objects and adaptors are defined in the header `<functional>`. Function objects that return Boolean values are known as predicates. The default binary predicate is the comparison `operator<`. In general, the elements being ordered need to be less than comparable so that, given any two elements, it can be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering among the nonequivalent elements.

Function templates

FUNCTION TEMPLATE	DESCRIPTION
<code>adjacent_find</code>	Searches for two adjacent elements that are either equal or satisfy a specified condition.
<code>all_of</code>	Returns true when a condition is present at each element in the given range.
<code>any_of</code>	Returns true when a condition is present at least once in the specified range of elements.
<code>binary_search</code>	Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.
<code>copy</code>	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.
<code>copy_backward</code>	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.
<code>copy_if</code>	Copy all elements in a given range that test true for a specified condition
<code>copy_n</code>	Copies a specified number of elements.
<code>count</code>	Returns the number of elements in a range whose values match a specified value.
<code>count_if</code>	Returns the number of elements in a range whose values match a specified condition.
<code>equal</code>	Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.
<code>equal_range</code>	Finds a pair of positions in an ordered range, the first less than or equivalent to the position of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.
<code>fill</code>	Assigns the same new value to every element in a specified range.
<code>fill_n</code>	Assigns a new value to a specified number of elements in a range starting with a particular element.
<code>find</code>	Locates the position of the first occurrence of an element in a range that has a specified value.

FUNCTION TEMPLATE	DESCRIPTION
<code>find_end</code>	Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.
<code>find_first_of</code>	Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.
<code>find_if</code>	Locates the position of the first occurrence of an element in a range that satisfies a specified condition.
<code>find_if_not</code>	Returns the first element in the indicated range that does not satisfy a condition.
<code>for_each</code>	Applies a specified function object to each element in a forward order within a range and returns the function object.
<code>generate</code>	Assigns the values generated by a function object to each element in a range.
<code>generate_n</code>	Assigns the values generated by a function object to a specified number of element in a range and returns to the position one past the last assigned value.
<code>includes</code>	Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.
<code>inplace_merge</code>	Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.
<code>is_heap</code>	Returns true if the elements in the specified range form a heap.
<code>is_heap_until</code>	Returns true if the specified range forms a heap until the last element.
<code>is_partitioned</code>	Returns true if all the elements in the given range that test true for a condition come before any elements that test false .
<code>is_permutation</code>	Determines whether the elements in a given range form a valid permutation.
<code>is_sorted</code>	Returns true if the elements in the specified range are in sorted order.
<code>is_sorted_until</code>	Returns true if the elements in the specified range are in sorted order.
<code>iter_swap</code>	Exchanges two values referred to by a pair of specified iterators.

FUNCTION TEMPLATE	DESCRIPTION
lexicographical_compare	Compares element by element between two sequences to determine which is lesser of the two.
lower_bound	Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value, where the ordering criterion may be specified by a binary predicate.
make_heap	Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.
max	Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.
max_element	Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.
merge	Combines all the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
min	Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.
min_element	Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.
minmax	Compares two input parameters and returns them as a pair, in order of least to greatest.
minmax_element	Performs the work performed by min_element and max_element in one call.
mismatch	Compares two ranges element by element either for equality or equivalent in a sense specified by a binary predicate and locates the first position where a difference occurs.
<alg> move	Move elements associated with a specified range.
move_backward	Moves the elements of one iterator to another. The move starts with the last element in a specified range, and ends with the first element in that range.
next_permutation	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
none_of	Returns true when a condition is never present among elements in the given range.

FUNCTION TEMPLATE	DESCRIPTION
<code>nth_element</code>	Partitions a range of elements, correctly locating the n th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.
<code>partial_sort</code>	Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.
<code>partial_sort_copy</code>	Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.
<code>partition</code>	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.
<code>partition_copy</code>	Copies elements for which a condition is true to one destination, and for which the condition is false to another. The elements must come from a specified range.
<code>partition_point</code>	Returns the first element in the given range that does not satisfy the condition. The elements are sorted so that those that satisfy the condition come before those that do not.
<code>pop_heap</code>	Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.
<code>prev_permutation</code>	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
<code>push_heap</code>	Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.
<code>random_shuffle</code>	Rearranges a sequence of N elements in a range into one of $N!$ possible arrangements selected at random.
<code>remove</code>	Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
<code>remove_copy</code>	Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.
<code>remove_copy_if</code>	Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

FUNCTION TEMPLATE	DESCRIPTION
remove_if	Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
replace	Examines each element in a range and replaces it if it matches a specified value.
replace_copy	Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.
replace_copy_if	Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.
replace_if	Examines each element in a range and replaces it if it satisfies a specified predicate.
reverse	Reverses the order of the elements within a range.
reverse_copy	Reverses the order of the elements within a source range while copying them into a destination range
rotate	Exchanges the elements in two adjacent ranges.
rotate_copy	Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.
search	Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.
search_n	Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.
set_difference	Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_intersection	Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_symmetric_difference	Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

FUNCTION TEMPLATE	DESCRIPTION
set_union	Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.
shuffle	Shuffles (rearranges) elements for a given range using a random number generator.
sort_heap	Converts a heap into a sorted range.
stable_partition	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.
stable_sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.
swap	Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.
swap_ranges	Exchanges the elements of one range with the elements of another, equal sized range.
transform	Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.
unique	Removes duplicate elements that are adjacent to each other in a specified range.
unique_copy	Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.
upper_bound	Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<algorithm> functions

1/16/2019 • 206 minutes to read • [Edit Online](#)

move	adjacent_find	all_of
any_of	binary_search	copy
copy_backward	copy_if	copy_n
count	count_if	equal
equal_range	fill	fill_n
find	find_end	find_first_of
find_if	find_if_not	for_each
generate	generate_n	includes
inplace_merge	is_heap	is_heap_until
is_partitioned	is_permutation	is_sorted
is_sorted_until	iter_swap	lexicographical_compare
lower_bound	make_heap	max
max_element	merge	min
min_element	minmax	minmax_element
mismatch	move_backward	next_permutation
none_of	nth_element	partial_sort
partial_sort_copy	partition	partition_copy
partition_point	pop_heap	prev_permutation
push_heap	random_shuffle	remove
remove_copy	remove_copy_if	remove_if
replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy

rotate	rotate_copy	search
search_n	set_difference	set_intersection
set_symmetric_difference	set_union	sort
sort_heap	stable_partition	stable_sort
shuffle	swap	swap_ranges
transform	unique	unique_copy
upper_bound		

adjacent_find

Searches for two adjacent elements that are either equal or satisfy a specified condition.

```
template<class ForwardIterator>
ForwardIterator adjacent_find(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate comp);
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched.

last

A forward iterator addressing the position one past the final element in the range to be searched.

comp

The binary predicate giving the condition to be satisfied by the values of the adjacent elements in the range being searched.

Return Value

A forward iterator to the first element of the adjacent pair that are either equal to each other (in the first version) or that satisfy the condition given by the binary predicate (in the second version), provided that such a pair of elements is found. Otherwise, an iterator pointing to *last* is returned.

Remarks

The `adjacent_find` algorithm is a nonmutating sequence algorithm. The range to be searched must be valid; all pointers must be dereferenceable and the last position is reachable from the first by incrementation. The time complexity of the algorithm is linear in the number of elements contained in the range.

The `operator==` used to determine the match between elements must impose an equivalence relation between its operands.

Example

```

// alg_adj_fnd.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

// Returns whether second element is twice the first
bool twice (int elem1, int elem2 )
{
    return elem1 * 2 == elem2;
}

int main()
{
    using namespace std;
    list<int> L;
    list<int>::iterator Iter;
    list<int>::iterator result1, result2;

    L.push_back( 50 );
    L.push_back( 40 );
    L.push_back( 10 );
    L.push_back( 20 );
    L.push_back( 20 );

    cout << "L = ( " ;
    for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
        cout << *Iter << " ";
    cout << ")" << endl;

    result1 = adjacent_find( L.begin( ), L.end( ) );
    if ( result1 == L.end( ) )
        cout << "There are not two adjacent elements that are equal."
            << endl;
    else
        cout << "There are two adjacent elements that are equal."
            << "\n They have a value of "
            << *( result1 ) << "." << endl;

    result2 = adjacent_find( L.begin( ), L.end( ), twice );
    if ( result2 == L.end( ) )
        cout << "There are not two adjacent elements where the "
            << " second is twice the first." << endl;
    else
        cout << "There are two adjacent elements where "
            << "the second is twice the first."
            << "\n They have values of " << *(result2++);
        cout << " & " << *result2 << "." << endl;
}

```

```

L = ( 50 40 10 20 20 )
There are two adjacent elements that are equal.
They have a value of 20.
There are two adjacent elements where the second is twice the first.
They have values of 10 & 20.

```

all_of

Returns **true** when a condition is present at each element in the given range.

```
template<class InputIterator, class Predicate>
bool all_of(
    InputIterator first,
    InputIterator last,
    BinaryPredicate comp);
```

Parameters

first

An input iterator that indicates where to start to check for a condition. The iterator marks where a range of elements starts.

last

An input iterator that indicates the end of the range of elements to check for a condition.

comp

A condition to test for. This is a user-defined predicate function object that defines the condition to be satisfied by an element being checked. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns **true** if the condition is detected at each element in the indicated range, and **false** if the condition is not detected at least one time.

Remarks

The template function returns **true** only if, for each N in the range $[0, \text{last} - \text{first})$, the predicate `comp(*(_First + N))` is **true**.

any_of

Returns **true** when a condition is present at least once in the specified range of elements.

```
template<class InputIterator, class UnaryPredicate>
bool any_of(
    InputIterator first,
    InputIterator last,
    UnaryPredicate comp);
```

Parameters

first

An input iterator that indicates where to start checking a range of elements for a condition.

last

An input iterator that indicates the end of the range of elements to check for a condition.

comp

A condition to test for. This is provided by a user-defined predicate function object. The predicate defines the condition to be satisfied by the element being tested. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns **true** if the condition is detected at least once in the indicated range, **false** if the condition is never detected.

Remarks

The template function returns **true** only if, for some N in the range

$[0, \text{last} - \text{first})$, the predicate `comp(*(first + N))` is true.

binary_search

Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.

```
template<class ForwardIterator, class Type>
bool binary_search(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ForwardIterator, class Type, class BinaryPredicate>
bool binary_search(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    BinaryPredicate comp);
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched.

last

A forward iterator addressing the position one past the final element in the range to be searched.

value

The value required to be matched by the value of the element or that must satisfy the condition with the element value specified by the binary predicate.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if an element is found in the range that is equal or equivalent to the specified value; otherwise, **false**.

Remarks

The sorted source range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The sorted range must each be arranged as a precondition to the application of the `binary_search` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The source ranges are not modified by `binary_search`.

The value types of the forward iterators need to be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to $(\text{last} - \text{first})$.

Example

```
// alg_bin_srch.cpp
// compile with: /EHsc
#include <list>
#include <vector>
#include <algorithm>
#include <functional> // greater<int>()
```

```

#include <functional> // greater<int>()
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    list<int> List1;

    List1.push_back( 50 );
    List1.push_back( 10 );
    List1.push_back( 30 );
    List1.push_back( 20 );
    List1.push_back( 25 );
    List1.push_back( 5 );

    List1.sort();

    cout << "List1 = ( " ;
    for ( auto Iter : List1 )
        cout << Iter << " ";
    cout << ")" << endl;

    // default binary search for 10
    if ( binary_search(List1.begin(), List1.end(), 10) )
        cout << "There is an element in list List1 with a value equal to 10."
        << endl;
    else
        cout << "There is no element in list List1 with a value equal to 10."
        << endl;

    // a binary_search under the binary predicate greater
    List1.sort(greater<int>());
    if ( binary_search(List1.begin(), List1.end(), 10, greater<int>()) )
        cout << "There is an element in list List1 with a value greater than 10 "
        << "under greater than." << endl;
    else
        cout << "No element in list List1 with a value greater than 10 "
        << "under greater than." << endl;

    // a binary_search under the user-defined binary predicate mod_lesser
    vector<int> v1;

    for ( auto i = -2; i <= 4; ++i )
    {
        v1.push_back(i);
    }

    sort(v1.begin(), v1.end(), mod_lesser);

    cout << "Ordered using mod_lesser, vector v1 = ( " ;
    for ( auto Iter : v1 )
        cout << Iter << " ";
    cout << ")" << endl;

    if ( binary_search(v1.begin(), v1.end(), -3, mod_lesser) )
        cout << "There is an element with a value equivalent to -3 "
        << "under mod_lesser." << endl;
    else
        cout << "There is no element with a value equivalent to -3 "
        << "under mod_lesser." << endl;
}

```



```
    cout << "There is not an element with a value equivalent to -3 "
    << "under mod_lesser." << endl;
}
```

copy

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator first,
    InputIterator last,
    OutputIterator destBeg);
```

Parameters

first

An input iterator addressing the position of the first element in the source range.

last

An input iterator addressing the position that is one past the final element in the source range.

destBeg

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses `result + (last - first)`.

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

Because the algorithm copies the source elements in order beginning with the first element, the destination range can overlap with the source range provided the *last* position of the source range is not contained in the destination range. `copy` can be used to shift elements to the left but not the right, unless there is no overlap between the source and destination ranges. To shift to the right any number of positions, use the [copy_backward](#) algorithm.

The `copy` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

Example

```

// alg_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1.push_back( 10 * i );

    int ii;
    for ( ii = 0 ; ii <= 10 ; ii++ )
        v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy the first 3 elements of v1 into the middle of v2
    copy( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 4 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the left
    copy( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 2 );

    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;
}

```

```

v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )

```

copy_backward

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 first,
    BidirectionalIterator1 last,
    BidirectionalIterator2 destEnd);
```

Parameters

first

A bidirectional iterator addressing the position of the first element in the source range.

last

A bidirectional iterator addressing the position that is one past the final element in the source range.

destEnd

A bidirectional iterator addressing the position of one past the final element in the destination range.

Return Value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses *destEnd* - (*last* - *first*).

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

The `copy_backward` algorithm imposes more stringent requirements than that the `copy` algorithm. Both its input and output iterators must be bidirectional.

The `copy_backward` and `move_backward` algorithms are the only C++ Standard Library algorithms designating the output range with an iterator pointing to the end of the destination range.

Because the algorithm copies the source elements in order beginning with the last element, the destination range can overlap with the source range provided the *first* position of the source range is not contained in the destination range. `copy_backward` can be used to shift elements to the right but not the left, unless there is no overlap between the source and destination ranges. To shift to the left any number of positions, use the `copy` algorithm.

The `copy_backward` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

Example

```

// alg_copy_bkwd.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 5 ; ++i )
        v1.push_back( 10 * i );

    int ii;
    for ( ii = 0 ; ii <= 10 ; ++ii )
        v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; ++Iter1 )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy_backward the first 3 elements of v1 into the middle of v2
    copy_backward( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 7 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the right
    copy_backward( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 9 );

    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;
}

```

copy_if

In a range of elements, copies the elements that are **true** for the specified condition.

```

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator copy_if(
    InputIterator first,
    InputIterator last,
    OutputIterator dest,
    Predicate pred);

```

Parameters

first

An input iterator that indicates the start of a range to check for the condition.

last

An input iterator that indicates the end of the range.

dest

The output iterator that indicates the destination for the copied elements.

_Pred

The condition against which every element in the range is tested. This condition is provided by a user-defined predicate function object. A predicate takes one argument and returns **true** or **false**.

Return Value

An output iterator that equals *dest* incremented once for each element that fulfills the condition. In other words, the return value minus *dest* equals the number of copied elements.

Remarks

The template function evaluates

```
if (pred(*_First + N) * dest++ = *(_First + N))
```

once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. If *dest* and *first* designate regions of storage, *dest* must not be in the range `[first, last)`.

copy_n

Copies a specified number of elements.

```
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(
    InputIterator first,
    Size count,
    OutputIterator dest);
```

Parameters

first

An input iterator that indicates where to copy elements from.

count

A signed or unsigned integer type specifying the number of elements to copy.

dest

An output iterator that indicates where to copy elements to.

Return Value

Returns an output iterator where elements have been copied to. It is the same as the returned value of the third parameter, *dest*.

Remarks

The template function evaluates `*(dest + N) = *(first + N)` once for each `N` in the range `[0, count)`, for strictly increasing values of `N` starting with the lowest value. It then returns `dest + N`. If *dest* and *first* designate regions of storage, *dest* must not be in the range `[first, last)`.

count

Returns the number of elements in a range whose values match a specified value.

```
template<class InputIterator, class Type>
typename iterator_traits<InputIterator>::difference_type count(
    InputIterator first,
    InputIterator last,
    const Type& val);
```

Parameters

first

An input iterator addressing the position of the first element in the range to be traversed.

last

An input iterator addressing the position one past the final element in the range to be traversed.

val

The value of the elements to be counted.

Return Value

The difference type of the `InputIterator` that counts the number of elements in the range `[first, last)` that have value *val*.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

This algorithm is generalized to count elements that satisfy any predicate with the template function [count_if](#).

Example

```
// alg_count.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result;
    result = count(v1.begin(), v1.end(), 10);
    cout << "The number of 10s in v2 is: " << result << "." << endl;
}
```

```
v1 = ( 10 20 10 40 10 )
The number of 10s in v2 is: 3.
```

count_if

Returns the number of elements in a range whose values satisfy a specified condition.

```
template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type count_if(
    InputIterator first,
    InputIterator last,
    Predicate pred);
```

Parameters

first

An input iterator addressing the position of the first element in the range to be searched.

last

An input iterator addressing the position one past the final element in the range to be searched.

_Pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be counted. A predicate takes single argument and returns **true** or **false**.

Return Value

The number of elements that satisfy the condition specified by the predicate.

Remarks

This template function is a generalization of the algorithm [count](#), replacing the predicate "equals a specific value" with any predicate.

Example

```

// alg_count_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater10(int value)
{
    return value > 10;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greater10);
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;
}

```

```

v1 = ( 10 20 10 40 10 )
The number of elements in v1 greater than 10 is: 2.

```

equal

Compares two ranges element by element for equality or equivalence in a sense specified by a binary predicate.

Use `std::equal` when comparing elements in different container types (for example `vector` and `list`) or when comparing different element types, or when you need to compare sub-ranges of containers. Otherwise, when comparing elements of the same type in the same container type, use the non-member `operator==` that is provided for each container.

Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range will not detect differences if the second range is longer than the first range, and will result in undefined behavior if the second range is shorter than the first range.


```

template<class InputIterator1, class InputIterator2>
bool equal(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    BinaryPredicate Comp); // C++14

template<class InputIterator1, class InputIterator2>
bool equal(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    InputIterator2 Last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    InputIterator2 Last2,
    BinaryPredicate Comp);

```

Parameters

First1

An input iterator addressing the position of the first element in the first range to be tested.

Last1

An input iterator addressing the position one past the last element in the first range to be tested.

First2

An input iterator addressing the position of the first element in the second range to be tested.

First2

An input iterator addressing the position of one past the last element in the second range to be tested.

Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if and only if the ranges are identical or equivalent under the binary predicate when compared element by element; otherwise, **false**.

Remarks

The range to be searched must be valid; all iterators must be dereferenceable and the last position is reachable from the first by incrementation.

If the two ranges are equal length, then the time complexity of the algorithm is linear in the number of elements contained in the range. Otherwise the function immediately returns **false**.

Neither the `operator==` nor the user-defined predicate is required to impose an equivalence relation that symmetric, reflexive and transitive between its operands.

Example

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v1 { 0, 5, 10, 15, 20, 25 };
    vector<int> v2 { 0, 5, 10, 15, 20, 25 };
    vector<int> v3 { 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };

    // Using range-and-a-half equal:
    bool b = equal(v1.begin(), v1.end(), v2.begin());
    cout << "v1 and v2 are equal: "
         << b << endl; // true, as expected

    b = equal(v1.begin(), v1.end(), v3.begin());
    cout << "v1 and v3 are equal: "
         << b << endl; // true, surprisingly

    // Using dual-range equal:
    b = equal(v1.begin(), v1.end(), v3.begin(), v3.end());
    cout << "v1 and v3 are equal with dual-range overload: "
         << b << endl; // false

    return 0;
}

```

equal_range

Given an ordered range, finds the subrange in which all elements are equivalent to a given value.

```

template<class ForwardIterator, class Type>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first,
    ForwardIterator last,
    const Type& val);

template<class ForwardIterator, class Type, class Predicate>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first,
    ForwardIterator last,
    const Type& val,
    Predicate comp);

```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched.

last

A forward iterator addressing the position one past the final element in the range to be searched.

val

The value being searched for in the ordered range.

comp

User-defined predicate function object that defines the sense in which one element is less than another.

Return Value

A pair of forward iterators that specify a subrange, contained within the range searched, in which all of the

elements are equivalent to *val* in the sense defined by the binary predicate used (either *comp* or the default, less-than).

If no elements in the range are equivalent to *val*, the returned pair of forward iterators are equal and specify the point where *val* could be inserted without disturbing the order of the range.

Remarks

The first iterator of the pair returned by the algorithm is [lower_bound](#), and the second iterator is [upper_bound](#).

The range must be sorted according to the predicate provided to `equal_range`. For example, if you are going to use the greater-than predicate, the range must be sorted in descending order.

Elements in the possibly empty subrange defined by the pair of iterators returned by `equal_range` will be equivalent to *val* in the sense defined by the predicate used.

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*last* - *first*).

Example

```
// alg_equal_range.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>()
#include <iostream>
#include <string>
using namespace std;

template<class T> void dump_vector( const vector<T>& v, pair<typename vector<T>::iterator, typename
vector<T>::iterator> range )
{
    // prints vector v with range delimited by [ and ]

    for ( typename vector<T>::const_iterator i = v.begin(); i != v.end(); ++i )
    {
        if ( i == range.first )
        {
            cout << "[ ";
        }
        if ( i == range.second )
        {
            cout << "] ";
        }

        cout << *i << " ";
    }
    cout << endl;
}

template<class T> void equal_range_demo( const vector<T>& original_vector, T val )
{
    vector<T> v(original_vector);

    sort( v.begin(), v.end() );
    cout << "Vector sorted by the default binary predicate <:" << endl << '\t';
    for ( vector<T>::const_iterator i = v.begin(); i != v.end(); ++i )
    {
        cout << *i << " ";
    }
    cout << endl << endl;

    pair<vector<T>::iterator, vector<T>::iterator> result
        = equal_range( v.begin(), v.end(), val );

    cout << "Result of equal_range with val = " << val << " " << endl << '\t';
```

```

    cout << "Result of equal_range with val = " << val << " : " << endl << '\t';
    dump_vector( v, result );
    cout << endl;
}

template<class T, class F> void equal_range_demo( const vector<T>& original_vector, T val, F pred, string
predname )
{
    vector<T> v(original_vector);

    sort( v.begin(), v.end(), pred );
    cout << "Vector sorted by the binary predicate " << predname << ":" << endl << '\t';
    for ( typename vector<T>::const_iterator i = v.begin(); i != v.end(); ++i )
    {
        cout << *i << " ";
    }
    cout << endl << endl;

    pair<typename vector<T>::iterator, typename vector<T>::iterator> result
        = equal_range( v.begin(), v.end(), val, pred );

    cout << "Result of equal_range with val = " << val << ":" << endl << '\t';
    dump_vector( v, result );
    cout << endl;
}

// Return whether absolute value of elem1 is less than absolute value of elem2
bool abs_lesser( int elem1, int elem2 )
{
    return abs(elem1) < abs(elem2);
}

// Return whether string l is shorter than string r
bool shorter_than(const string& l, const string& r)
{
    return l.size() < r.size();
}

int main()
{
    vector<int> v1;

    // Constructing vector v1 with default less than ordering
    for ( int i = -1; i <= 4; ++i )
    {
        v1.push_back(i);
    }

    for ( int i = -3; i <= 0; ++i )
    {
        v1.push_back( i );
    }

    equal_range_demo( v1, 3 );
    equal_range_demo( v1, 3, greater<int>(), "greater" );
    equal_range_demo( v1, 3, abs_lesser, "abs_lesser" );

    vector<string> v2;

    v2.push_back("cute");
    v2.push_back("fluffy");
    v2.push_back("kittens");
    v2.push_back("fun");
    v2.push_back("meowmeowmeow");
    v2.push_back("blah");

    equal_range_demo<string>( v2, "fred" );
    equal_range_demo<string>( v2, "fred", shorter_than, "shorter_than" );
}

```

fill

Assigns the same new value to every element in a specified range.

```
template<class ForwardIterator, class Type>
void fill(
    ForwardIterator first,
    ForwardIterator last,
    const Type& val);
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be traversed.

last

A forward iterator addressing the position one past the final element in the range to be traversed.

val

The value to be assigned to elements in the range [*first*, *last*).

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

Example

```
// alg_fill.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Fill the last 5 positions with a value of 2
    fill( v1.begin( ) + 5, v1.end( ), 2 );

    cout << "Modified v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )  
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

fill_n

Assigns a new value to a specified number of elements in a range beginning with a particular element.

```
template<class OutputIterator, class Size, class Type>  
OutputIterator fill_n(  
    OutputIterator First,  
    Size Count,  
    const Type& Val);
```

Parameters

First

An output iterator addressing the position of the first element in the range to be assigned the value *Val*.

Count

A signed or unsigned integer type specifying the number of elements to be assigned the value.

Val

The value to be assigned to elements in the range [*First*, *First* + *Count*).

Return Value

An iterator to the element that follows the last element filled if *Count* > zero, otherwise the first element.

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

Example

```

// alg_fill_n.cpp
// compile using /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v;

    for ( auto i = 0 ; i < 9 ; ++i )
        v.push_back( 0 );

    cout << " vector v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;

    // Fill the first 3 positions with a value of 1, saving position.
    auto pos = fill_n( v.begin(), 3, 1 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;

    // Fill the next 3 positions with a value of 2, using last position.
    fill_n( pos, 3, 2 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;

    // Fill the last 3 positions with a value of 3, using relative math.
    fill_n( v.end()-3, 3, 3 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;
}

```

find

Locates the position of the first occurrence of an element in a range that has a specified value.

```

template<class InputIterator, class T>
InputIterator find(
    InputIterator first,
    InputIterator last,
    const T& val);

```

Parameters

first

An input iterator addressing the position of the first element in the range to be searched for the specified value.

last

An input iterator addressing the position one past the final element in the range to be searched for the specified value.

val

The value to be searched for.

Return Value

An input iterator addressing the first occurrence of the specified value in the range being searched. If no element is found with an equivalent value, returns *last*.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

For a code example using `find()`, see [find_if](#).

find_end

Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2,
    ForwardIterator2 Last2);

template<class ForwardIterator1, class ForwardIterator2, class Pred>
ForwardIterator1 find_end(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2,
    ForwardIterator2 Last2,
    Pred Comp);
```

Parameters

First1

A forward iterator addressing the position of the first element in the range to be searched.

Last1

A forward iterator addressing the position one past the last element in the range to be searched.

First2

A forward iterator addressing the position of the first element in the range to search for.

Last2

A forward iterator addressing the position one past the last element in the range to search for.

Comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the last subsequence within [First1, Last1) that matches the specified sequence [First2, Last2).

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

```
// alg_find_end.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

int main()
{
    using namespace std;
    vector <int> v1, v2;
    list <int> L1;
    vector <int>::iterator Iter1, Iter2;
    list <int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
    for ( ii = 1 ; ii <= 4 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Searching v1 for a match to L1 under identity
    vector <int>::iterator result1;
    result1 = find_end ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

    if ( result1 == v1.end( ) )
```

```

    if ( result1 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a match of L1 in v1 that begins at "
            << "position " << result1 - v1.begin( ) << "." << endl;

    // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    result2 = find_end ( v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

    if ( result2 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a sequence of elements in v1 that "
            << "are equivalent to those\n in v2 under the binary "
            << "predicate twice and that begins at position "
            << result2 - v1.begin( ) << "." << endl;
}

```

```

Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 5 10 15 20 )
Vector v2 = ( 20 30 40 )
There is a match of L1 in v1 that begins at position 7.
There is a sequence of elements in v1 that are equivalent to those
in v2 under the binary predicate twice and that begins at position 8.

```

find_first_of

Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.

```

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1,
    ForwardIterator1 Last1,
    ForwardIterator2 first2,
    ForwardIterator2 Last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1,
    ForwardIterator1 Last1,
    ForwardIterator2 first2,
    ForwardIterator2 Last2,
    BinaryPredicate comp);

```

Parameters

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the final element in the range to be searched.

first2

A forward iterator addressing the position of the first element in the range to be matched.

last2

A forward iterator addressing the position one past the final element in the range to be matched.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

```
// alg_find_first_of.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

int main()
{
    using namespace std;
    vector <int> v1, v2;
    list <int> L1;
    vector <int>::iterator Iter1, Iter2;
    list <int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
    for ( ii = 3 ; ii <= 4 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
```

```

cout << "List L1 = ( " ;
for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
    cout << *L1_Iter << " ";
cout << ")" << endl;

cout << "Vector v2 = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")" << endl;

// Searching v1 for first match to L1 under identity
vector<int>::iterator result1;
result1 = find_first_of ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

if ( result1 == v1.end( ) )
    cout << "There is no match of L1 in v1."
        << endl;
else
    cout << "There is at least one match of L1 in v1"
        << "\n and the first one begins at "
        << "position "<< result1 - v1.begin( ) << "." << endl;

// Searching v1 for a match to L1 under the binary predicate twice
vector<int>::iterator result2;
result2 = find_first_of ( v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

if ( result2 == v1.end( ) )
    cout << "There is no match of L1 in v1."
        << endl;
else
    cout << "There is a sequence of elements in v1 that "
        << "are equivalent\n to those in v2 under the binary "
        << "predicate twice\n and the first one begins at position "
        << result2 - v1.begin( ) << "." << endl;
}

```

```

Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 15 20 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
and the first one begins at position 3.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.

```

find_if

Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

```

template<class InputIterator, class Predicate>
InputIterator find_if(
    InputIterator first,
    InputIterator last,
    Predicate pred);

```

Parameters

first

An input iterator addressing the position of the first element in the range to be searched.

last

An input iterator addressing the position one past the final element in the range to be searched.

pred

User-defined predicate function object or [lambda expression](#) that defines the condition to be satisfied by the element being searched for. A predicate takes single argument and returns **true** (satisfied) or **false** (not satisfied). The signature of *pred* must effectively be `bool pred(const T& arg);`, where `T` is a type to which `InputIterator` can be implicitly converted when dereferenced. The **const** keyword is shown only to illustrate that the function object or lambda should not modify the argument.

Return Value

An input iterator that refers to the first element in the range that satisfies the condition specified by the predicate (the predicate results in **true**). If no element is found to satisfy the predicate, returns *last*.

Remarks

This template function is a generalization of the algorithm [find](#), replacing the predicate "equals a specific value" with any predicate. For the logical opposite (find the first element that does not satisfy the predicate), see [find_if_not](#).

Example

```
// cl.exe /W4 /nologo /EHsc /MTd
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

template <typename S> void print(const S& s) {
    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }
    cout << endl;
}

// Test std::find()
template <class InputIterator, class T>
void find_print_result(InputIterator first, InputIterator last, const T& value) {

    // call <algorithm> std::find()
    auto p = find(first, last, value);

    cout << "value " << value;
    if (p == last) {
        cout << " not found." << endl;
    } else {
        cout << " found." << endl;
    }
}

// Test std::find_if()
template <class InputIterator, class Predicate>
void find_if_print_result(InputIterator first, InputIterator last,
    Predicate Pred, const string& Str) {

    // call <algorithm> std::find_if()
    auto p = find_if(first, last, Pred);

    if (p == last) {
        cout << Str << " not found." << endl;
    } else {
        cout << "first " << Str << " found: " << *p << endl;
    }
}

// Function to use as the UnaryPredicate argument to find_if() in this example
```

```

bool is_odd_int(int i) {
    return ((i % 2) != 0);
}

int main()
{
    // Test using a plain old array.
    const int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    cout << "array x[] contents: ";
    print(x);
    // Using non-member std::begin()/std::end() to get input iterators for the plain old array.
    cout << "Test std::find() with array..." << endl;
    find_print_result(begin(x), end(x), 10);
    find_print_result(begin(x), end(x), 42);
    cout << "Test std::find_if() with array..." << endl;
    find_if_print_result(begin(x), end(x), is_odd_int, "odd integer"); // function name
    find_if_print_result(begin(x), end(x), // lambda
        [](int i){ return ((i % 2) == 0); }, "even integer");

    // Test using a vector.
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back((i + 1) * 10);
    }
    cout << endl << "vector v contents: ";
    print(v);
    cout << "Test std::find() with vector..." << endl;
    find_print_result(v.begin(), v.end(), 20);
    find_print_result(v.begin(), v.end(), 12);
    cout << "Test std::find_if() with vector..." << endl;
    find_if_print_result(v.begin(), v.end(), is_odd_int, "odd integer");
    find_if_print_result(v.begin(), v.end(), // lambda
        [](int i){ return ((i % 2) == 0); }, "even integer");
}

```

find_if_not

Returns the first element in the indicated range that does not satisfy a condition.

```

template<class InputIterator, class Predicate>
InputIterator find_if_not(
    InputIterator first,
    InputIterator last,
    Predicate pred);

```

Parameters

first

An input iterator addressing the position of the first element in the range to be searched.

last

An input iterator addressing the position one past the final element in the range to be searched.

pred

User-defined predicate function object or [lambda expression](#) that defines the condition to be not satisfied by the element being searched for. A predicate takes single argument and returns **true** (satisfied) or **false** (not satisfied).

The signature of *pred* must effectively be `bool pred(const T& arg);`, where `T` is a type to which `InputIterator` can be implicitly converted when dereferenced. The **const** keyword is shown only to illustrate that the function object or lambda should not modify the argument.

Return Value

An input iterator that refers to the first element in the range that does not satisfy the condition specified by the

predicate (the predicate results in **false**). If all elements satisfy the predicate (the predicate results in **true** for every element), returns *last*.

Remarks

This template function is a generalization of the algorithm [find](#), replacing the predicate "equals a specific value" with any predicate. For the logical opposite (find the first element that does satisfy the predicate), see [find_if](#).

For a code example readily adaptable to `find_if_not()`, see [find_if](#).

for_each

Applies a specified function object to each element in a forward order within a range and returns the function object.

```
template<class InputIterator, class Function>
Function for_each(
    InputIterator first,
    InputIterator last,
    Function func);
```

Parameters

first

An input iterator addressing the position of the first element in the range to be operated on.

last

An input iterator addressing the position one past the final element in the range operated on.

_Func

User-defined function object that is applied to each element in the range.

Return Value

A copy of the function object after it has been applied to all of the elements in the range.

Remarks

The algorithm `for_each` is very flexible, allowing the modification of each element within a range in different, user-specified ways. Templated functions may be reused in a modified form by passing different parameters. User-defined functions may accumulate information within an internal state that the algorithm may return after processing all of the elements in the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear with at most (*last* - *first*) comparisons.

Example

```
// alg_for_each.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
    Type Factor;    // The value to multiply by
public:
```

```

// Constructor initializes the value to multiply by
MultValue ( const Type& val ) : Factor ( val ) {
}

// The function call for the element to be multiplied
void operator( ) ( Type& elem ) const
{
    elem *= Factor;
}
};

// The function object to determine the average
class Average
{
private:
    long num;        // The number of elements
    long sum;        // The sum of the elements
public:
    // Constructor initializes the value to multiply by
    Average( ) : num ( 0 ) , sum ( 0 )
    {
    }

    // The function call to process the next element
    void operator( ) ( int elem ) \
    {
        num++;        // Increment the element count
        sum += elem;   // Add the value to the partial sum
    }

    // return Average
    operator double( )
    {
        return static_cast <double> (sum) /
            static_cast <double> (num);
    }
};

int main()
{
    using namespace std;
    vector <int> v1;
    vector <int>::iterator Iter1;

    // Constructing vector v1
    int i;
    for ( i = -4 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Original vector  v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Using for_each to multiply each element by a Factor
    for_each ( v1.begin( ), v1.end( ), MultValue<int> ( -2 ) );

    cout << "Multiplying the elements of the vector v1\n "
        << "by the factor -2 gives:\n v1mod1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // The function object is templatized and so can be
    // used again on the elements with a different Factor
    for_each (v1.begin( ), v1.end( ), MultValue<int> (5 ) );

```



```

cout << "Multiplying the elements of the vector v1mod\n "
    << "by the factor 5 gives:\n v1mod2 = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

// The local state of a function object can accumulate
// information about a sequence of actions that the
// return value can make available, here the Average
double avemod2 = for_each ( v1.begin( ), v1.end( ),
    Average( ) );
cout << "The average of the elements of v1 is:\n Average ( v1mod2 ) = "
    << avemod2 << "." << endl;
}

```

```

Original vector  v1 = ( -4 -3 -2 -1 0 1 2 ).
Multiplying the elements of the vector v1
by the factor -2 gives:
v1mod1 = ( 8 6 4 2 0 -2 -4 ).
Multiplying the elements of the vector v1mod
by the factor 5 gives:
v1mod2 = ( 40 30 20 10 0 -10 -20 ).
The average of the elements of v1 is:
Average ( v1mod2 ) = 10.

```

generate

Assigns the values generated by a function object to each element in a range.

```

template<class ForwardIterator, class Generator>
void generate(
    ForwardIterator first,
    ForwardIterator last,
    Generator _Gen);

```

Parameters

first

A forward iterator addressing the position of the first element in the range to which values are to be assigned.

last

A forward iterator addressing the position one past the final element in the range to which values are to be assigned.

_Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterators for the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly (`last` - `first`) calls to the generator being required.

Example

```

// alg_generate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

int main()
{
    using namespace std;

    // Assigning random values to vector integer elements
    vector<int> v1 ( 5 );
    vector<int>::iterator Iter1;
    deque<int> deq1 ( 5 );
    deque<int>::iterator d1_Iter;

    generate ( v1.begin( ), v1.end( ), rand );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Assigning random values to deque integer elements
    generate ( deq1.begin( ), deq1.end( ), rand );

    cout << "Deque deq1 is ( " ;
    for ( d1_Iter = deq1.begin( ) ; d1_Iter != deq1.end( ) ; d1_Iter++ )
        cout << *d1_Iter << " ";
    cout << ")." << endl;
}

```

```

Vector v1 is ( 41 18467 6334 26500 19169 ).
Deque deq1 is ( 15724 11478 29358 26962 24464 ).

```

generate_n

Assigns the values generated by a function object to a specified number of elements in a range and returns to the position one past the last assigned value.

```

template<class OutputIterator, class Diff, class Generator>
void generate_n(
    OutputIterator First,
    Diff Count,
    Generator Gen);

```

Parameters

First

An output iterator addressing the position of first element in the range to which values are to be assigned.

Count

A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.

Gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterators for the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly `Count` calls to the generator being required.

Example

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <vector>
#include <deque>
#include <iostream>
#include <string>
#include <algorithm>
#include <random>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    const int elemcount = 5;
    vector<int> v(elemcount);
    deque<int> dq(elemcount);

    // Set up random number distribution
    random_device rd;
    mt19937 engine(rd());
    uniform_int_distribution<int> dist(-9, 9);

    // Call generate_n, using a lambda for the third parameter
    generate_n(v.begin(), elemcount, [&]() { return dist(engine); });
    print("vector v is: ", v);

    generate_n(dq.begin(), elemcount, [&]() { return dist(engine); });
    print("deque dq is: ", dq);
}
```

includes

Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2>
bool includes(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool includes(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the first sorted range contains all the elements in the second sorted range; otherwise, **false**.

Remarks

Another way to think of this test is that it determined whether the second source range is a subset of the first source range.

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

The sorted source ranges must each be arranged as a precondition to the application of the algorithm `includes` in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The source ranges are not modified by the algorithm `merge`.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. More precisely, the algorithm tests whether all the elements in the first sorted range under a specified binary predicate have equivalent ordering to those in the second sorted range.

The complexity of the algorithm is linear with at most $2 * ((last1 - first1) - (*last2 - first2*)) - 1$ comparisons for

nonempty source ranges.

Example

```
// alg_includes.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector <int> v1a, v1b;
    vector <int>::iterator Iter1a, Iter1b;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -2 ; i <= 4 ; i++ )
    {
        v1a.push_back( i );
    }

    int ii;
    for ( ii = -2 ; ii <= 3 ; ii++ )
    {
        v1b.push_back( ii );
    }

    cout << "Original vector v1a with range sorted by the\n "
        << "binary predicate less than is v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
        << "binary predicate less than is v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vectors v2a & v2b with ranges sorted by greater
    vector <int> v2a ( v1a ) , v2b ( v1b );
    vector <int>::iterator Iter2a, Iter2b;
    sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
    sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );
    v2a.pop_back( );

    cout << "Original vector v2a with range sorted by the\n "
        << "binary predicate greater is v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
        << "binary predicate greater is v2b = ( " ;
    for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
```

```

        cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
    vector<int> v3a ( v1a ), v3b ( v1b );
    vector<int>::iterator Iter3a, Iter3b;
    reverse (v3a.begin( ), v3a.end( ) );
    v3a.pop_back( );
    v3a.pop_back( );
    sort ( v3a.begin( ), v3a.end( ), mod_lesser );
    sort ( v3b.begin( ), v3b.end( ), mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
        << "binary predicate mod_lesser is v3a = ( " ;
    for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
        cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
        << "binary predicate mod_lesser is v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To test for inclusion under an ascending order
    // with the default binary predicate less<int>( )
    bool Result1;
    Result1 = includes ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ) );
    if ( Result1 )
        cout << "All the elements in vector v1b are "
            << "contained in vector v1a." << endl;
    else
        cout << "At least one of the elements in vector v1b "
            << "is not contained in vector v1a." << endl;

    // To test for inclusion under descending
    // order specify binary predicate greater<int>( )
    bool Result2;
    Result2 = includes ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ), greater<int>( ) );
    if ( Result2 )
        cout << "All the elements in vector v2b are "
            << "contained in vector v2a." << endl;
    else
        cout << "At least one of the elements in vector v2b "
            << "is not contained in vector v2a." << endl;

    // To test for inclusion under a user
    // defined binary predicate mod_lesser
    bool Result3;
    Result3 = includes ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), mod_lesser );
    if ( Result3 )
        cout << "All the elements in vector v3b are "
            << "contained under mod_lesser in vector v3a."
            << endl;
    else
        cout << "At least one of the elements in vector v3b is "
            << " not contained under mod_lesser in vector v3a."
            << endl;
}

```

Original vector v1a with range sorted by the
binary predicate less than is v1a = (-2 -1 0 1 2 3 4).
Original vector v1b with range sorted by the
binary predicate less than is v1b = (-2 -1 0 1 2 3).
Original vector v2a with range sorted by the
binary predicate greater is v2a = (4 3 2 1 0 -1).
Original vector v2b with range sorted by the
binary predicate greater is v2b = (3 2 1 0 -1 -2).
Original vector v3a with range sorted by the
binary predicate mod_less is v3a = (0 1 2 3 4).
Original vector v3b with range sorted by the
binary predicate mod_less is v3b = (0 -1 1 -2 2 3).
All the elements in vector v1b are contained in vector v1a.
At least one of the elements in vector v2b is not contained in vector v2a.
At least one of the elements in vector v3b is not contained under mod_less in vector v3a.

inplace_merge

Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.

```
template<class BidirectionalIterator>
void inplace_merge(
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last);

template<class BidirectionalIterator, class Predicate>
void inplace_merge(
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last,
    Predicate comp);
```

Parameters

first

A bidirectional iterator addressing the position of the first element in the first of two consecutive sorted ranges to be combined and sorted into a single range.

middle

A bidirectional iterator addressing the position of the first element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

last

A bidirectional iterator addressing the position one past the last element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Remarks

The sorted consecutive ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

The sorted consecutive ranges must each be arranged as a precondition to the application of the `inplace_merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges. The

operation is stable as the relative order of elements within each range is preserved. When there are equivalent elements in both source ranges, the element from the first range precedes the element from the second in the combined range.

The complexity depends on the available memory as the algorithm allocates memory to a temporary buffer. If sufficient memory is available, the best case is linear with $(last - first) - 1$ comparisons; if no auxiliary memory is available, the worst case is $N \log(N)$, where $N = (last - first)$.

Example

```
// alg_inplace_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      //For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector <int> v1;
    vector <int>::iterator Iter1, Iter2, Iter3;

    // Constructing vector v1 with default less-than ordering
    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = -5 ; ii <= 0 ; ii++ )
    {
        v1.push_back( ii );
    }

    cout << "Original vector v1 with subranges sorted by the\n "
        << "binary predicate less than is  v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Constructing vector v2 with ranges sorted by greater
    vector <int> v2 ( v1 );
    vector <int>::iterator break2;
    break2 = find ( v2.begin( ), v2.end( ), -5 );
    sort ( v2.begin( ), break2 , greater<int>( ) );
    sort ( break2 , v2.end( ), greater<int>( ) );
    cout << "Original vector v2 with subranges sorted by the\n "
        << "binary predicate greater is v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Constructing vector v3 with ranges sorted by mod_lesser
    vector <int> v3 ( v1 );
    vector <int>::iterator break3;
```



```

break3 = find ( v3.begin( ), v3.end( ), -5 );
sort ( v3.begin( ), break3 , mod_lesser );
sort ( break3 , v3.end( ), mod_lesser );
cout << "Original vector v3 with subranges sorted by the\n "
    << "binary predicate mod_lesser is v3 = ( " ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")" << endl;

vector<int>::iterator break1;
break1 = find (v1.begin( ), v1.end( ), -5 );
inplace_merge ( v1.begin( ), break1, v1.end( ) );
cout << "Merged inplace with default order,\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")" << endl;

// To Merge inplace in descending order, specify binary
// predicate greater<int>( )
inplace_merge ( v2.begin( ), break2 , v2.end( ) , greater<int>( ) );
cout << "Merged inplace with binary predicate greater specified,\n "
    << "vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")" << endl;

// Applying a user defined (UD) binary predicate mod_lesser
inplace_merge ( v3.begin( ), break3, v3.end( ), mod_lesser );
cout << "Merged inplace with binary predicate mod_lesser specified,\n "
    << "vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")" << endl;
}

```

```

Original vector v1 with subranges sorted by the
binary predicate less than is v1 = ( 0 1 2 3 4 5 -5 -4 -3 -2 -1 0 )
Original vector v2 with subranges sorted by the
binary predicate greater is v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Original vector v3 with subranges sorted by the
binary predicate mod_lesser is v3 = ( 0 1 2 3 4 5 0 -1 -2 -3 -4 -5 )
Merged inplace with default order,
vector v1mod = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 )
Merged inplace with binary predicate greater specified,
vector v2mod = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Merged inplace with binary predicate mod_lesser specified,
vector v3mod = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 )

```

is_heap

Returns **true** if the elements in the specified range form a heap.

```

template<class RandomAccessIterator>
bool is_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class BinaryPredicate>
bool is_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    BinaryPredicate comp);

```

Parameters

first

A random access iterator that indicates the start of a range to check for a heap.

last

A random access iterator that indicates the end of a range.

comp

A condition to test to order elements. A binary predicate takes a single argument and returns **true** or **false**.

Return Value

Returns **true** if the elements in the specified range form a heap, **false** if they do not.

Remarks

The first template function returns `is_heap_until` (`first` , `last`) == `last` .

The second template function returns

```
is_heap_until(first, last, comp) == last .
```

is_heap_until

Returns an iterator positioned at the first element in the range [`begin` , `end`) that does not satisfy the heap ordering condition, or *end* if the range forms a heap.

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(
    RandomAccessIterator begin,
    RandomAccessIterator end);

template<class RandomAccessIterator, class BinaryPredicate>
RandomAccessIterator is_heap_until(
    RandomAccessIterator begin,
    RandomAccessIterator end,
    BinaryPredicate compare);
```

Parameters

begin

A random access iterator that specifies the first element of a range to check for a heap.

end

A random access iterator that specifies the end of the range to check for a heap.

compare

A binary predicate that specifies the strict weak ordering condition that defines a heap. The default predicate when *compare* is not specified is `std::less<>` .

Return Value

Returns *end* if the specified range forms a heap or contains one or fewer elements. Otherwise, returns an iterator for the first element found that does not satisfy the heap condition.

Remarks

The first template function returns the last iterator `next` in [`begin` , `end`] where [`begin` , `next`) is a heap ordered by the function object `std::less<>` . If the distance `end - begin < 2` , the function returns *end*.

The second template function behaves the same as the first, except that it uses the predicate `compare` instead of `std::less<>` as the heap ordering condition.

is_partitioned

Returns **true** if all the elements in the given range that test **true** for a condition come before any elements that test **false**.

```
template<class InputIterator, class BinaryPredicate>
bool is_partitioned(
    InputIterator first,
    InputIterator last,
    BinaryPredicate comp);
```

Parameters

first

An input iterator that indicates where a range starts to check for a condition.

last

An input iterator that indicates the end of a range.

comp

The condition to test for. This is provided by a user-defined predicate function object that defines the condition to be satisfied by the element being searched for. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns true when all of the elements in the given range that test **true** for a condition come before any elements that test **false**, and otherwise returns **false**.

Remarks

The template function returns **true** only if all elements in `[first , last)` are partitioned by *comp*; that is, all elements `x` in `[first , last)` for which `comp (x)` is true occur before all elements `y` for which `comp (y)` is **false**.

is_permutation

Returns true if both ranges contain the same elements, whether or not the elements are in the same order. Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range will not detect differences if the second range is longer than the first range, and will result in undefined behavior if the second range is shorter than the first range.

```

template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2);

template<class ForwardIterator1, class ForwardIterator2, class Predicate>
bool is_permutation(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2,
    Predicate Pred);

// C++14
template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2,
    ForwardIterator2 Last2);

template<class ForwardIterator1, class ForwardIterator2, class Predicate>
bool is_permutation(
    ForwardIterator1 First1,
    ForwardIterator1 Last1,
    ForwardIterator2 First2,
    ForwardIterator2 Last2,
    Predicate Pred);

```

Parameters

First1

A forward iterator that refers to the first element of the range.

Last1

A forward iterator that refers one past the last element of the range.

First2

A forward iterator that refers to the first element of a second range, used for comparison.

Last2

A forward iterator that refers to one past the last element of a second range, used for comparison.

Pred

A predicate that tests for equivalence and returns a **bool**.

Return Value

true when the ranges can be rearranged so as to be identical according to the comparator predicate; otherwise, **false**.

Remarks

`is_permutation` has quadratic complexity in the worst case.

The first template function assumes that there are as many elements in the range beginning at *First2* as there are in the range designated by [`First1`, `Last1`). If there are more elements in the second range, they are ignored; if there are less, undefined behavior will occur. The third template function (C++14 and later) does not make this assumption. Both return **true** only if, for each element *X* in the range designated by [`First1`, `Last1`) there are as many elements *Y* in the same range for which *X* == *Y* as there are in the range beginning at *First2* or [`First2`, `Last2`). Here, `operator==` must perform a pairwise comparison between its operands.

The second and fourth template functions behave the same, except that they replace `operator==(X, Y)` with

`Pred(X, Y)` . To behave correctly, the predicate must be symmetric, reflexive and transitive.

Example

The following example shows how to use `is_permutation` :

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;

int main()
{
    vector<int> vec_1{ 2, 3, 0, 1, 4, 5 };
    vector<int> vec_2{ 5, 4, 0, 3, 1, 2 };

    vector<int> vec_3{ 4, 9, 13, 3, 6, 5 };
    vector<int> vec_4{ 7, 4, 11, 9, 2, 1 };

    cout << "(1) Compare using built-in == operator: ";
    cout << boolalpha << is_permutation(vec_1.begin(), vec_1.end(),
        vec_2.begin(), vec_2.end()) << endl; // true

    cout << "(2) Compare after modifying vec_2: ";
    vec_2[0] = 6;
    cout << is_permutation(vec_1.begin(), vec_1.end(),
        vec_2.begin(), vec_2.end()) << endl; // false

    // Define equivalence as "both are odd or both are even"
    cout << "(3) vec_3 is a permutation of vec_4: ";
    cout << is_permutation(vec_3.begin(), vec_3.end(),
        vec_4.begin(), vec_4.end(),
        [](int lhs, int rhs) { return lhs % 2 == rhs % 2; }) << endl; // true

    // Initialize a vector using the 's' string literal to specify a std::string
    vector<string> animals_1{ "dog"s, "cat"s, "bird"s, "monkey"s };
    vector<string> animals_2{ "donkey"s, "bird"s, "meerkat"s, "cat"s };

    // Define equivalence as "first letters are equal":
    bool is_perm = is_permutation(animals_1.begin(), animals_1.end(), animals_2.begin(), animals_2.end(),
        [](const string& lhs, const string& rhs)
        {
            return lhs[0] == rhs[0]; //std::string guaranteed to have at least a null terminator
        });

    cout << "animals_2 is a permutation of animals_1: " << is_perm << endl; // true

    cout << "Press a letter" << endl;
    char c;
    cin >> c;

    return 0;
}
```

is_sorted

Returns **true** if the elements in the specified range are in sorted order.

```

template<class ForwardIterator>
bool is_sorted(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
bool is_sorted(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate comp);

```

Parameters

first

A forward iterator that indicates where the range to check begins.

last

A forward iterator that indicates the end of a range.

comp

The condition to test to determine an order between two elements. A predicate takes a single argument and returns **true** or **false**. This performs the same task as `operator<`.

Remarks

The first template function returns `is_sorted_until(first, last) == last`. The `operator<` function performs the order comparison.

The second template function returns `is_sorted_until(first, last , comp) == last`. The `comp` predicate function performs the order comparison.

is_sorted_until

Returns a `ForwardIterator` that is set to the last element that is in sorted order from a specified range.

The second version lets you provide a `BinaryPredicate` function that returns **true** when two given elements are in sorted order, and **false** otherwise.

```

template<class ForwardIterator>
ForwardIterator is_sorted_until(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator is_sorted_until(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate comp);

```

Parameters

first

A forward iterator that indicates where the range to check starts.

last

A forward iterator that indicates the end of a range.

comp

The condition to test to determine an order between two elements. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns a `ForwardIterator` set to the last element in sorted order. The sorted sequence starts from *first*.

Remarks

The first template function returns the last iterator `next` in `[first , last]` so that `[first , next)` is a sorted sequence ordered by `operator<`. If `distance()` `< 2` the function returns *last*.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `comp (X, Y)`.

iter_swap

Exchanges two values referred to by a pair of specified iterators.

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap( ForwardIterator1 left, ForwardIterator2 right );
```

Parameters

left

One of the forward iterators whose value is to be exchanged.

right

The second of the forward iterators whose value is to be exchanged.

Remarks

`swap` should be used in preference to `iter_swap`, which was included in the C++ Standard for backward compatibility. If `Fit1` and `Fit2` are forward iterators, then `iter_swap (Fit1 , Fit2)`, is equivalent to `swap (* Fit1 , * Fit2)`.

The value types of the input forward iterators must have the same value.

Example

```
// alg_iter_swap.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){ }
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){ }
    CInt& operator=( const CInt& rhs ) { m_nVal =
        rhs.m_nVal; return *this; }
    bool operator<( const CInt& rhs ) const
        { return ( m_nVal < rhs.m_nVal ); }
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt(" << rhs.m_nVal << ")";
```

```

    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    CInt c1 = 5, c2 = 1, c3 = 10;
    deque<CInt> deq1;
    deque<CInt>::iterator d1_Iter;

    deq1.push_back ( c1 );
    deq1.push_back ( c2 );
    deq1.push_back ( c3 );

    cout << "The original deque of CInts is deq1 = (" ;
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ", ";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Exchanging first and last elements with iter_swap
    iter_swap ( deq1.begin( ), --deq1.end( ) );

    cout << "The deque of CInts with first & last elements swapped is:\n deq1 = (" ;
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ", ";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Swapping back first and last elements with swap
    swap ( *deq1.begin( ), *(deq1.end( ) - 1 ) );

    cout << "The deque of CInts with first & last elements swapped back is:\n deq1 = (" ;
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ", ";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Swapping a vector element with a deque element
    vector<int> v1;
    vector<int>::iterator Iter1;
    deque<int> deq2;
    deque<int>::iterator d2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 4 ; ii <= 5 ; ii++ )
    {
        deq2.push_back( ii );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ); Iter1 != v1.end( ); Iter1++ )
        cout << *Iter1 << " ";
    cout << " )." << endl;

```



```

cout << "Deque deq2 is ( " ;
for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
    cout << *d2_Iter << " ";
cout << ")." << endl;

iter_swap ( v1.begin( ), deq2.begin( ) );

cout << "After exchanging first elements,\n vector v1 is: v1 = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl << " & deque deq2 is: deq2 = ( " ;
for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
    cout << *d2_Iter << " ";
cout << ")." << endl;
}

```

The original deque of CInts is deq1 = (CInt(5), CInt(1), CInt(10)).
 The deque of CInts with first & last elements swapped is:
 deq1 = (CInt(10), CInt(1), CInt(5)).
 The deque of CInts with first & last elements swapped back is:
 deq1 = (CInt(5), CInt(1), CInt(10)).
 Vector v1 is (0 1 2 3).
 Deque deq2 is (4 5).
 After exchanging first elements,
 vector v1 is: v1 = (4 1 2 3).
 & deque deq2 is: deq2 = (0 5).

lexicographical_compare

Compares element by element between two sequences to determine which is lesser of the two.

```

template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2 );

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool lexicographical_compare(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first range to be compared.

last1

An input iterator addressing the position one past the final element in the first range to be compared.

first2

An input iterator addressing the position of the first element in the second range to be compared.

last2

An input iterator addressing the position one past the final element in the second range to be compared.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the first range is lexicographically less than the second range; otherwise **false**.

Remarks

A lexicographical comparison between sequences compares them element by element until:

- It finds two corresponding elements unequal, and the result of their comparison is taken as the result of the comparison between sequences.
- No inequalities are found, but one sequence has more elements than the other, and the shorter sequence is considered less than the longer sequence.
- No inequalities are found and the sequences have the same number of elements, and so the sequences are equal and the result of the comparison is false.

Example

```
// alg_lex_comp.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 < elem2;
}

int main()
{
    using namespace std;
    vector <int> v1, v2;
    list <int> L1;
    vector <int>::iterator Iter1, Iter2;
    list <int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    int ii;
    for ( ii = 0 ; ii <= 6 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
```

```

        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Self lexicographical_comparison of v1 under identity
    bool result1;
    result1 = lexicographical_compare (v1.begin( ), v1.end( ),
                                       v1.begin( ), v1.end( ) );
    if ( result1 )
        cout << "Vector v1 is lexicographically_less than v1." << endl;
    else
        cout << "Vector v1 is not lexicographically_less than v1." << endl;

    // lexicographical_comparison of v1 and L2 under identity
    bool result2;
    result2 = lexicographical_compare (v1.begin( ), v1.end( ),
                                       L1.begin( ), L1.end( ) );
    if ( result2 )
        cout << "Vector v1 is lexicographically_less than L1." << endl;
    else
        cout << "Vector v1 is lexicographically_less than L1." << endl;

    bool result3;
    result3 = lexicographical_compare (v1.begin( ), v1.end( ),
                                       v2.begin( ), v2.end( ), twice );
    if ( result3 )
        cout << "Vector v1 is lexicographically_less than v2 "
              << "under twice." << endl;
    else
        cout << "Vector v1 is not lexicographically_less than v2 "
              << "under twice." << endl;
}

```

```

Vector v1 = ( 0 5 10 15 20 25 )
List L1 = ( 0 5 10 15 20 25 30 )
Vector v2 = ( 0 10 20 30 40 50 )
Vector v1 is not lexicographically_less than v1.
Vector v1 is lexicographically_less than L1.
Vector v1 is not lexicographically_less than v2 under twice.

```

lower_bound

Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value, where the ordering criterion may be specified by a binary predicate.

```

template<class ForwardIterator, class Type>
ForwardIterator lower_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value );

template<class ForwardIterator, class Type, class BinaryPredicate>
ForwardIterator lower_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    BinaryPredicate comp );

```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched.

last

A forward iterator addressing the position one past the final element in the range to be searched.

value

The value whose first position or possible first position is being searched for in the ordered range.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator at the position of the first element in an ordered range with a value that is greater than or equivalent to a specified value, where the equivalence is specified with a binary predicate.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of using `lower_bound` and where the ordering is the same as specified by with binary predicate.

The range is not modified by the algorithm `lower_bound`.

The value types of the forward iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to $(\text{last} - \text{first})$.

Example

```
// alg_lower_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    vector<int> v1;
    // Constructing vector v1 with default less-than ordering
    for ( auto i = -1 ; i <= 4 ; ++i )
    {
        v1.push_back( i );
    }
}
```

```

for ( auto ii =-3 ; ii <= 0 ; ++ii )
{
    v1.push_back( ii );
}

cout << "Starting vector v1 = ( " ;
for (const auto &Iter : v1)
    cout << Iter << " ";
cout << ")." << endl;

sort(v1.begin(), v1.end());
cout << "Original vector v1 with range sorted by the\n "
    << "binary predicate less than is v1 = ( " ;
for (const auto &Iter : v1)
    cout << Iter << " ";
cout << ")." << endl;

// Constructing vector v2 with range sorted by greater
vector<int> v2(v1);

sort(v2.begin(), v2.end(), greater<int>());

cout << "Original vector v2 with range sorted by the\n "
    << "binary predicate greater is v2 = ( " ;
for (const auto &Iter : v2)
    cout << Iter << " ";
cout << ")." << endl;

// Constructing vectors v3 with range sorted by mod_lesser
vector<int> v3(v1);
sort(v3.begin(), v3.end(), mod_lesser);

cout << "Original vector v3 with range sorted by the\n "
    << "binary predicate mod_lesser is v3 = ( " ;
for (const auto &Iter : v3)
    cout << Iter << " ";
cout << ")." << endl;

// Demonstrate lower_bound

vector<int>::iterator Result;

// lower_bound of 3 in v1 with default binary predicate less<int>()
Result = lower_bound(v1.begin(), v1.end(), 3);
cout << "The lower_bound in v1 for the element with a value of 3 is: "
    << *Result << "." << endl;

// lower_bound of 3 in v2 with the binary predicate greater<int>( )
Result = lower_bound(v2.begin(), v2.end(), 3, greater<int>());
cout << "The lower_bound in v2 for the element with a value of 3 is: "
    << *Result << "." << endl;

// lower_bound of 3 in v3 with the binary predicate mod_lesser
Result = lower_bound(v3.begin(), v3.end(), 3, mod_lesser);
cout << "The lower_bound in v3 for the element with a value of 3 is: "
    << *Result << "." << endl;
}

```

make_heap

Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.

```

template<class RandomAccessIterator>
void make_heap(
    RandomAccessIterator first,
    RandomAccessIterator last );

template<class RandomAccessIterator, class BinaryPredicate>
void make_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    BinaryPredicate comp );

```

Parameters

first

A random-access iterator addressing the position of the first element in the range to be converted into a heap.

last

A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the C++ Standard Library container adaptor [priority_queue Class](#).

The complexity is linear, requiring $3 * (*last - first)$ comparisons.

Example

```

// alg_make_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with default less than ordering
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with greater than ordering
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater-than heaped version of v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

max

Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.

```

template<class Type>
constexpr Type& max(
    const Type& left,
    const Type& right);
template<class Type, class Pr>
constexpr Type& max(
    const Type& left,
    const Type& right,
    BinaryPredicate comp);
template<class Type>
constexpr Type& max (
    initializer_list<Type> );
template<class Type, class Pr>
constexpr Type& max(
    initializer_list<Type> ,
    BinaryPredicate comp);

```

Parameters

left

The first of the two objects being compared.

right

The second of the two objects being compared.

comp

A binary predicate used to compare the two objects.

_lList

The initializer list that contains the objects to be compared.

Return Value

The greater of the two objects, unless neither is greater; in that case, it returns the first of the two objects. In the case of an initializer_list, it returns the greatest of the objects in the list.

Remarks

The `max` algorithm is unusual in having objects passed as parameters. Most C++ Standard Library algorithms operate on a range of elements whose position is specified by iterators passed as parameters. If you need a function that operates on a range of elements, use `max_element` instead. Visual Studio 2017 enables **constexpr** on the overloads that take an initializer_list.

Example

```
// alg_max.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<< ( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
        rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<< ( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<< ( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether absolute value of elem1 is greater than
// absolute value of elem2
bool abs_greater ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = -elem1;
    if ( elem2 < 0 )
        elem2 = -elem2;
```



```

    return elem1 < elem2;
};

int main()
{
    int a = 6, b = -7;
    // Return the integer with the larger absolute value
    const int& result1 = max(a, b, abs_greater);
    // Return the larger integer
    const int& result2 = max(a, b);

    cout << "Using integers 6 and -7..." << endl;
    cout << "The integer with the greater absolute value is: "
        << result1 << "." << endl;
    cout << "The integer with the greater value is: "
        << result2 << "." << endl;
    cout << endl;

    // Comparing the members of an initializer_list
    const int& result3 = max({ a, b });
    const int& result4 = max({ a, b }, abs_greater);

    cout << "Comparing the members of an initializer_list..." << endl;
    cout << "The member with the greater value is: " << result3 << endl;
    cout << "The integer with the greater absolute value is: " << result4 << endl;

    // Comparing set containers with elements of type CInt
    // using the max algorithm
    CInt c1 = 1, c2 = 2, c3 = 3;
    set<CInt> s1, s2, s3;
    set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s2.insert ( c2 );
    s2.insert ( c3 );

    cout << "s1 = (" ;
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << "," ;
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    cout << "s2 = (" ;
    for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
        cout << " " << *s2_Iter << "," ;
    s2_Iter = --s2.end( );
    cout << " " << *s2_Iter << " )." << endl;

    s3 = max ( s1, s2 );
    cout << "s3 = max ( s1, s2 ) = (" ;
    for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
        cout << " " << *s3_Iter << "," ;
    s3_Iter = --s3.end( );
    cout << " " << *s3_Iter << " )." << endl << endl;

    // Comparing vectors with integer elements using the max algorithm
    vector<int> v1, v2, v3, v4, v5;
    vector<int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

    int i;
    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 0 ; ii <= 2 ; ii++ )
    {

```

```

        v2.push_back( ii );
    }

    int iii;
    for ( iii = 0 ; iii <= 2 ; iii++ )
    {
        v3.push_back( 2 * iii );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    cout << "Vector v3 is ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;

    v4 = max ( v1, v2 );
    v5 = max ( v1, v3 );

    cout << "Vector v4 = max (v1,v2) is ( " ;
    for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
        cout << *Iter4 << " ";
    cout << ")." << endl;

    cout << "Vector v5 = max (v1,v3) is ( " ;
    for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
        cout << *Iter5 << " ";
    cout << ")." << endl;
}

```

Using integers 6 and -7...

The integer with the greater absolute value is: -7

The integer with the greater value is: 6.

Comparing the members of an initializer_list...The member with the greater value is: 6The integer wiht the greater absolute value is: -7

```

s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = max ( s1, s2 ) = ( CInt( 2 ), CInt( 3 ) ).

```

Vector v1 is (0 1 2).

Vector v2 is (0 1 2).

Vector v3 is (0 2 4).

Vector v4 = max (v1,v2) is (0 1 2).

Vector v5 = max (v1,v3) is (0 2 4).

max_element

Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.

```

template<class ForwardIterator>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last );

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last, BinaryPredicate comp );

```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched for the largest element.

last

A forward iterator addressing the position one past the final element in the range to be searched for the largest element.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

A forward iterator addressing the position of the first occurrence of the largest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: $(\text{last} - \text{first}) - 1$ comparisons are required for a nonempty range.

Example

```
// alg_max_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<< ( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
        rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<< ( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is greater than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
```

```

    return elem1 < elem2;
};

int main()
{
    // Searching a set container with elements of type CInt
    // for the maximum element
    CInt c1 = 1, c2 = 2, c3 = -3;
    set<CInt> s1;
    set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s1.insert ( c3 );

    cout << "s1 = (" ;
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ",";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    s1_R1_Iter = max_element ( s1.begin( ), s1.end( ) );

    cout << "The largest element in s1 is: " << *s1_R1_Iter << endl;
    cout << endl;

    // Searching a vector with elements of type int for the maximum
    // element under default less than & mod_lesser binary predicates
    vector<int> v1;
    vector<int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 1 ; ii <= 4 ; ii++ )
    {
        v1.push_back( - 2 * ii );
    }

    cout << "Vector v1 is ( " ;
    for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
        cout << *v1_Iter << " ";
    cout << " )." << endl;

    v1_R1_Iter = max_element ( v1.begin( ), v1.end( ) );
    v1_R2_Iter = max_element ( v1.begin( ), v1.end( ), mod_lesser);

    cout << "The largest element in v1 is: " << *v1_R1_Iter << endl;
    cout << "The largest element in v1 under the mod_lesser"
        << "\n binary predicate is: " << *v1_R2_Iter << endl;
}

```

merge

Combines all of the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate>
OutputIterator merge(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be combined and sorted into a single range.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm `merge`.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range.

The complexity of the algorithm is linear with at most $(last1 - first1) - (last2 - first2) - 1$ comparisons.

The [list class](#) provides a member function "merge" to merge the elements of two lists.

Example

```
// alg_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional> // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 ) {
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main() {
    using namespace std;
    vector <int> v1a, v1b, v1 ( 12 );
    vector <int>::iterator Iter1a, Iter1b, Iter1;

    // Constructing vector v1a and v1b with default less than ordering
    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1a.push_back( i );

    int ii;
    for ( ii = -5 ; ii <= 0 ; ii++ )
        v1b.push_back( ii );

    cout << "Original vector v1a with range sorted by the\n "
        << "binary predicate less than is  v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
        << "binary predicate less than is  v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vector v2 with ranges sorted by greater
    vector <int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector <int>::iterator Iter2a, Iter2b, Iter2;
    sort ( v2a.begin( ) , v2a.end( ) , greater<int>( ) );
    sort ( v2b.begin( ) , v2b.end( ) , greater<int>( ) );

    cout << "Original vector v2a with range sorted by the\n "
        << "binary predicate greater is  v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
```

```

cout << "Original vector v2b with range sorted by the\n "
    << "binary predicate greater is   v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vector v3 with ranges sorted by mod_lesser
vector<int> v3a ( v1a ), v3b ( v1b ), v3 ( v1 );
vector<int>::iterator Iter3a, Iter3b, Iter3;
sort ( v3a.begin( ), v3a.end( ), mod_lesser );
sort ( v3b.begin( ), v3b.end( ), mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
    << "binary predicate mod_lesser is   v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
    << "binary predicate mod_lesser is   v3b = ( " ;
for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
    cout << *Iter3b << " ";
cout << ")." << endl;

// To merge inplace in ascending order with default binary
// predicate less<int>( )
merge ( v1a.begin( ), v1a.end( ), v1b.begin( ), v1b.end( ), v1.begin( ) );
cout << "Merged inplace with default order,\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

// To merge inplace in descending order, specify binary
// predicate greater<int>( )
merge ( v2a.begin( ), v2a.end( ), v2b.begin( ), v2b.end( ),
    v2.begin( ), greater<int>( ) );
cout << "Merged inplace with binary predicate greater specified,\n "
    << "vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")." << endl;

// Applying A user-defined (UD) binary predicate mod_lesser
merge ( v3a.begin( ), v3a.end( ), v3b.begin( ), v3b.end( ),
    v3.begin( ), mod_lesser );
cout << "Merged inplace with binary predicate mod_lesser specified,\n "
    << "vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")." << endl;
}

```

min

Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.

```

template<class Type>
constexpr const Type& min(
    const Type& left,
    const Type& right);
template<class Type, class Pr>
constexpr const Type& min(
    const Type& left,
    const Type& right,
    BinaryPredicate comp);
template<class Type>
constexpr Type min(
    initializer_list<Type> );
template<class Type, class Pr>
constexpr Type min(
    initializer_list<Type>,
    BinaryPredicate comp);

```

Parameters

left

The first of the two objects being compared.

right

The second of the two objects being compared.

comp

A binary predicate used to compare the two objects.

_lList

The initializer_list that contains the members to be compared.

Return Value

The lesser of the two objects, unless neither is lesser; in that case, it returns the first of the two objects. In the case of an initializer_list, it returns the least of the objects in the list.

Remarks

The `min` algorithm is unusual in having objects passed as parameters. Most C++ Standard Library algorithms operate on a range of elements whose position is specified by iterators passed as parameters. If you need a function that uses a range of elements, use [min_element](#). `constexpr` was enabled on the `initializer_list` overloads in Visual Studio 2017.

Example

```

// alg_min.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<< ( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
        rhs.m_nVal; return *this;}
    bool operator/ ( const CInt& rhs ) const

```



```

bool operator<( const CInt& rhs ) const
{
    return ( m_nVal < rhs.m_nVal );
}

friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Comparing integers directly using the min algorithm with
    // binary predicate mod_lesser & with default less than
    int a = 6, b = -7, c = 7 ;
    const int& result1 = min ( a, b, mod_lesser );
    const int& result2 = min ( b, c );

    cout << "The mod_lesser of the integers 6 & -7 is: "
        << result1 << "." << endl;
    cout << "The lesser of the integers -7 & 7 is: "
        << result2 << "." << endl;
    cout << endl;

    // Comparing the members of an initializer_list
    const int& result3 = min({ a, c });
    const int& result4 = min({ a, b }, mod_lesser);

    cout << "The lesser of the integers 6 & 7 is: "
        << result3 << "." << endl;
    cout << "The mod_lesser of the integers 6 & -7 is: "
        << result4 << "." << endl;
    cout << endl;

    // Comparing set containers with elements of type CInt
    // using the min algorithm
    CInt c1 = 1, c2 = 2, c3 = 3;
    set<CInt> s1, s2, s3;
    set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s2.insert ( c2 );
    s2.insert ( c3 );

    cout << "s1 = (" ;
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ", ";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    cout << "s2 = (" ;
    for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
        cout << " " << *s2_Iter << ", ";
    s2_Iter = --s2.end( );
    cout << " " << *s2_Iter << " )." << endl;
}

```

```

cout << " " << *s2_Iter << " )." << endl;

s3 = min ( s1, s2 );
cout << "s3 = min ( s1, s2 ) = (";
for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
    cout << " " << *s3_Iter << ",";
s3_Iter = --s3.end( );
cout << " " << *s3_Iter << " )." << endl << endl;

// Comparing vectors with integer elements using min algorithm
vector<int> v1, v2, v3, v4, v5;
vector<int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

int i;
for ( i = 0 ; i <= 2 ; i++ )
{
    v1.push_back( i );
}

int ii;
for ( ii = 0 ; ii <= 2 ; ii++ )
{
    v2.push_back( ii );
}

int iii;
for ( iii = 0 ; iii <= 2 ; iii++ )
{
    v3.push_back( 2 * iii );
}

cout << "Vector v1 is ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

cout << "Vector v2 is ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")." << endl;

cout << "Vector v3 is ( " ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")." << endl;

v4 = min ( v1, v2 );
v5 = min ( v1, v3 );

cout << "Vector v4 = min ( v1,v2 ) is ( " ;
for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
    cout << *Iter4 << " ";
cout << ")." << endl;

cout << "Vector v5 = min ( v1,v3 ) is ( " ;
for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
    cout << *Iter5 << " ";
cout << ")." << endl;
}

```

```

The mod_lesser of the integers 6 & -7 is: 6.
The lesser of the integers -7 & 7 is: -7.
The lesser of the integers 6 & 7 is: 6.The mod_lesser of the integers 6 & -7 is: 6.
s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = min ( s1, s2 ) = ( CInt( 1 ), CInt( 2 ) ).

Vector v1 is ( 0 1 2 ).
Vector v2 is ( 0 1 2 ).
Vector v3 is ( 0 2 4 ).
Vector v4 = min ( v1,v2 ) is ( 0 1 2 ).
Vector v5 = min ( v1,v3 ) is ( 0 1 2 ).

```

min_element

Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.

```

template<class ForwardIterator>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last );

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator min_element(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate comp);

```

Parameters

first

A forward iterator addressing the position of the first element in the range to be searched for the smallest element.

last

A forward iterator addressing the position one past the final element in the range to be searched for the smallest element.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

A forward iterator addressing the position of the first occurrence of the smallest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: $(\text{last} - \text{first}) - 1$ comparisons are required for a nonempty range.

Example

```

// alg_min_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

```

```

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){ }
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){ }
    CInt& operator=( const CInt& rhs ) {m_nVal =
        rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Searching a set container with elements of type CInt
    // for the minimum element
    CInt c1 = 1, c2 = 2, c3 = -3;
    set<CInt> s1;
    set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s1.insert ( c3 );

    cout << "s1 = (" ;
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ",";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    s1_R1_Iter = min_element ( s1.begin( ), s1.end( ) );

    cout << "The smallest element in s1 is: " << *s1_R1_Iter << endl;
    cout << endl;

    // Searching a vector with elements of type int for the maximum
    // element under default less than & mod_lesser binary predicates
    vector<int> v1;
    vector<int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }
}

```

```

int ii;
for ( ii = 1 ; ii <= 4 ; ii++ )
{
    v1.push_back( - 2 * ii );
}

cout << "Vector v1 is ( " ;
for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
    cout << *v1_Iter << " ";
cout << ")." << endl;

v1_R1_Iter = min_element ( v1.begin( ), v1.end( ) );
v1_R2_Iter = min_element ( v1.begin( ), v1.end( ), mod_lesser);

cout << "The smallest element in v1 is: " << *v1_R1_Iter << endl;
cout << "The smallest element in v1 under the mod_lesser"
    << "\n binary predicate is: " << *v1_R2_Iter << endl;
}

```

```

s1 = ( CInt( -3 ), CInt( 1 ), CInt( 2 ) ).
The smallest element in s1 is: CInt( -3 )

Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 ).
The smallest element in v1 is: -8
The smallest element in v1 under the mod_lesser
binary predicate is: 0

```

minmax_element

Performs the work performed by `min_element` and `max_element` in one call.

```

template<class ForwardIterator>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first,
    ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate comp);

```

Parameters

first

A forward iterator that indicates the beginning of a range.

last

A forward iterator that indicates the end of a range.

comp

An optional test used to order elements.

Return Value

Returns

```
pair<ForwardIterator, ForwardIterator>
```

```
( min_element (first, last), max_element (first, last)) .
```

Remarks

The first template function returns

```
pair<ForwardIterator, ForwardIterator>
```

```
(min_element(_First, Last), max_element(_First, Last)) .
```

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `comp (X, Y)` .

If the sequence is non-empty, the function performs at most $3 * (last - first - 1) / 2$ comparisons.

minmax

Compares two input parameters and returns them as a pair, in order of lesser to greater.

```
template<class Type>
constexpr pair<const Type&, const Type&> minmax(
    const Type& left,
    const Type& right);
template<class Type, class BinaryPredicate>
constexpr pair<const Type&, const Type&> minmax(
    const Type& left,
    const Type& right,
    BinaryPredicate comp);
template<class Type>
constexpr pair<Type&, Type&> minmax(
    initializer_list<Type> );
template<class Type, class BinaryPredicate>
constexpr pair<Type&, Type&> minmax(
    initializer_list<Type>,
    BinaryPredicate comp);
```

Parameters

left

The first of the two objects being compared.

right

The second of the two objects being compared.

comp

A binary predicate used to compare the two objects.

_lList

The initializer_list that contains the members to be compared.

Remarks

The first template function returns `pair<const Type&, const Type&>(right , left)` if *right* is less than *left*.

Otherwise, it returns `pair<const Type&, const Type&>(left , right)` .

The second member function returns a pair where the first element is the lesser and the second is the greater when compared by the predicate *comp*.

The remaining template functions behave the same, except that they replace the *left* and *right* parameters with *_lList*.

The function performs exactly one comparison.

mismatch

Compares two ranges element by element and locates the first position where a difference occurs.

Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range will not detect differences if the second range is longer than the first range, and will result in undefined behavior if the second range is shorter than the first range.

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2 );

template<class InputIterator1, class InputIterator2, class BinaryPredicate> pair<InputIterator1,
InputIterator2>>
mismatch(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    BinaryPredicate Comp );

//C++14
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    InputIterator2 Last2 );

template<class InputIterator1, class InputIterator2, class BinaryPredicate> pair<InputIterator1,
InputIterator2>>
mismatch(
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    InputIterator2 Last2,
    BinaryPredicate Comp);
```

Parameters

First1

An input iterator addressing the position of the first element in the first range to be tested.

Last1

An input iterator addressing the position one past the last element in the first range to be tested.

First2

An input iterator addressing the position of the first element in the second range to be tested.

Last2

An input iterator addressing the position of one past the last element in the second range to be tested.

Comp

User-defined predicate function object that compares the current elements in each range and determines whether they are equivalent. It returns **true** when satisfied and **false** when not satisfied.

Return Value

A pair of iterators addressing the positions of the mismatch in the two ranges, the first component iterator to the position in the first range and the second component iterator to the position in the second range. If there is no difference between the elements in the ranges compared or if the binary predicate in the second version is satisfied by all element pairs from the two ranges, then the first component iterator points to the position one past the final element in the first range and the second component iterator to position one past the final element tested in the second range.

Remarks

The first template function assumes that there are as many elements in the range beginning at first2 as there are in the range designated by [first1, last1). If there are more in the second range, they are ignored; if there are less then undefined behavior will result.

The range to be searched must be valid; all iterators must be dereferenceable and the last position is reachable from the first by incrementation.

The time complexity of the algorithm is linear in the number of elements contained in the shorter range.

The user-defined predicate is not required to impose an equivalence relation that symmetric, reflexive and transitive between its operands.

Example

The following example demonstrates how to use mismatch. The C++03 overload is shown only in order to demonstrate how it can produce an unexpected result.

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>

using namespace std;

// Return whether first element is twice the second
// Note that this is not a symmetric, reflexive and transitive equivalence.
// mismatch and equal accept such predicates, but is_permutation does not.
bool twice(int elem1, int elem2)
{
    return elem1 == elem2 * 2;
}

void PrintResult(const string& msg, const pair<vector<int>::iterator, vector<int>::iterator>& result,
    const vector<int>& left, const vector<int>& right)
{
    // If either iterator stops before reaching the end of its container,
    // it means a mismatch was detected.
    if (result.first != left.end() || result.second != right.end())
    {
        string leftpos(result.first == left.end() ? "end" : to_string(*result.first));
        string rightpos(result.second == right.end() ? "end" : to_string(*result.second));
        cout << msg << "mismatch. Left iterator at " << leftpos
            << " right iterator at " << rightpos << endl;
    }
    else
    {
        cout << msg << " match." << endl;
    }
}

int main()
{
    vector<int> vec_1{ 0, 5, 10, 15, 20, 25 };
    vector<int> vec_2{ 0, 5, 10, 15, 20, 25, 30 };

    // Testing different length vectors for mismatch (C++03)
    auto match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin());
    bool is_mismatch = match_vecs.first != vec_1.end();
    cout << "C++03: vec_1 and vec_2 are a mismatch: " << boolalpha << is_mismatch << endl;

    // Using dual-range overloads:
```



```

// Testing different length vectors for mismatch (C++14)
match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin(), vec_2.end());
PrintResult("C++14: vec_1 and vec_2: ", match_vecs, vec_1, vec_2);

// Identify point of mismatch between vec_1 and modified vec_2.
vec_2[3] = 42;
match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin(), vec_2.end());
PrintResult("C++14 vec_1 v. vec_2 modified: ", match_vecs, vec_1, vec_2);

// Test vec_3 and vec_4 for mismatch under the binary predicate twice (C++14)
vector<int> vec_3{ 10, 20, 30, 40, 50, 60 };
vector<int> vec_4{ 5, 10, 15, 20, 25, 30 };
match_vecs = mismatch(vec_3.begin(), vec_3.end(), vec_4.begin(), vec_4.end(), twice);
PrintResult("vec_3 v. vec_4 with pred: ", match_vecs, vec_3, vec_4);

vec_4[5] = 31;
match_vecs = mismatch(vec_3.begin(), vec_3.end(), vec_4.begin(), vec_4.end(), twice);
PrintResult("vec_3 v. modified vec_4 with pred: ", match_vecs, vec_3, vec_4);

// Compare a vector<int> to a list<int>
list<int> list_1{ 0, 5, 10, 15, 20, 25 };
auto match_vec_list = mismatch(vec_1.begin(), vec_1.end(), list_1.begin(), list_1.end());
is_mismatch = match_vec_list.first != vec_1.end() || match_vec_list.second != list_1.end();
cout << "vec_1 and list_1 are a mismatch: " << boolalpha << is_mismatch << endl;

char c;
cout << "Press a key" << endl;
cin >> c;

}

/*
Output:
C++03: vec_1 and vec_2 are a mismatch: false
C++14: vec_1 and vec_2: mismatch. Left iterator at end right iterator at 30
C++14 vec_1 v. vec_2 modified: mismatch. Left iterator at 15 right iterator at 42
C++14 vec_3 v. vec_4 with pred: match.
C++14 vec_3 v. modified vec_4 with pred: mismatch. Left iterator at 60 right iterator at 31
C++14: vec_1 and list_1 are a mismatch: false
Press a key
*/

```

<alg> move

Move elements associated with a specified range.

```

template<class InputIterator, class OutputIterator>
OutputIterator move(
    InputIterator first,
    InputIterator last,
    OutputIterator dest);

```

Parameters

first

An input iterator that indicates where to start the range of elements to move.

last

An input iterator that indicates the end of a range of elements to move.

dest

The output iterator that is to contain the moved elements.

Remarks

The template function evaluates `*(dest + N) = move(*(first + N))` once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. It then returns `dest + N`. If `dest` and `first` designate regions of storage, `dest` must not be in the range `[first, last)`.

move_backward

Moves the elements of one iterator to another. The move starts with the last element in a specified range, and ends with the first element in that range.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
    BidirectionalIterator2 move_backward(
        BidirectionalIterator1 first,
        BidirectionalIterator1 last,
        BidirectionalIterator2 destEnd);
```

Parameters

first

An iterator that indicates the start of a range to move elements from.

last

An iterator that indicates the end of a range to move elements from. This element is not moved.

destEnd

A bidirectional iterator addressing the position of one past the final element in the destination range.

Remarks

The template function evaluates `*(destEnd - N - 1) = move(*(last - N - 1))` once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. It then returns `destEnd - (last - first)`. If `destEnd` and `first` designate regions of storage, `destEnd` must not be in the range `[first, last)`.

`move` and `move_backward` are functionally equivalent to using `copy` and `copy_backward` with a move iterator.

next_permutation

Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.

```
template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class BinaryPredicate>
    bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, BinaryPredicate comp);
```

Parameters

first

A bidirectional iterator pointing to the position of the first element in the range to be permuted.

last

A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the lexicographically next permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically smallest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less than comparable to insure that the next permutation is well defined.

The complexity is linear with at most $(\text{last} - \text{first})/2$ swaps.

Example

```
// alg_next_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
    { return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Reordering the elements of type CInt in a deque
    // using the prev_permutation algorithm
    CInt c1 = 5, c2 = 1, c3 = 10;
    bool deq1Result;
    deque<CInt> deq1, deq2, deq3;
    deque<CInt>::iterator d1_Iter;
```

```

deq1.push_back ( c1 );
deq1.push_back ( c2 );
deq1.push_back ( c3 );

cout << "The original deque of CInts is deq1 = (" ;
for ( d1_Iter = deq1.begin( ) ; d1_Iter != --deq1.end( ) ; d1_Iter++ )
    cout << " " << *d1_Iter << ", ";
d1_Iter = --deq1.end( ) ;
cout << " " << *d1_Iter << " )." << endl;

deq1Result = next_permutation ( deq1.begin( ), deq1.end( ) );

if ( deq1Result )
    cout << "The lexicographically next permutation "
        << "exists and has\nreplaced the original "
        << "ordering of the sequence in deq1." << endl;
else
    cout << "The lexicographically next permutation doesn't "
        << "exist\n and the lexicographically "
        << "smallest permutation\n has replaced the "
        << "original ordering of the sequence in deq1." << endl;

cout << "After one application of next_permutation,\n deq1 = (" ;
for ( d1_Iter = deq1.begin( ) ; d1_Iter != --deq1.end( ) ; d1_Iter++ )
    cout << " " << *d1_Iter << ", ";
d1_Iter = --deq1.end( ) ;
cout << " " << *d1_Iter << " )." << endl << endl;

// Permuting vector elements with binary function mod_lesser
vector <int> v1;
vector <int>::iterator Iter1;

int i;
for ( i = -3 ; i <= 3 ; i++ )
{
    v1.push_back( i );
}

cout << "Vector v1 is ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << " )." << endl;

next_permutation ( v1.begin( ), v1.end( ), mod_lesser );

cout << "After the first next_permutation, vector v1 is:\n v1 = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << " )." << endl;

int iii = 1;
while ( iii <= 5 ) {
    next_permutation ( v1.begin( ), v1.end( ), mod_lesser );
    cout << "After another next_permutation of vector v1,\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << " )." << endl;
    iii++;
}
}

```

The original deque of CInts is `deq1 = (CInt(5), CInt(1), CInt(10))`.
The lexicographically next permutation exists and has replaced the original ordering of the sequence in `deq1`.
After one application of `next_permutation`,
`deq1 = (CInt(5), CInt(10), CInt(1))`.

Vector `v1` is `(-3 -2 -1 0 1 2 3)`.
After the first `next_permutation`, vector `v1` is:
`v1 = (-3 -2 -1 0 1 3 2)`.
After another `next_permutation` of vector `v1`,
`v1 = (-3 -2 -1 0 2 1 3)`.
After another `next_permutation` of vector `v1`,
`v1 = (-3 -2 -1 0 2 3 1)`.
After another `next_permutation` of vector `v1`,
`v1 = (-3 -2 -1 0 3 1 2)`.
After another `next_permutation` of vector `v1`,
`v1 = (-3 -2 -1 0 3 2 1)`.
After another `next_permutation` of vector `v1`,
`v1 = (-3 -2 -1 1 0 2 3)`.

nth_element

Partitions a range of elements, correctly locating the *n*th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.

```
template<class RandomAccessIterator>
void nth_element( RandomAccessIterator first, RandomAccessIterator _Nth, RandomAccessIterator last);

template<class RandomAccessIterator, class BinaryPredicate>
void nth_element( RandomAccessIterator first, RandomAccessIterator _Nth, RandomAccessIterator last,
BinaryPredicate comp);
```

Parameters

first

A random-access iterator addressing the position of the first element in the range to be partitioned.

_Nth

A random-access iterator addressing the position of element to be correctly ordered on the boundary of the partition.

last

A random-access iterator addressing the position one past the final element in the range to be partitioned.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The `nth_element` algorithm does not guarantee that elements in the sub-ranges either side of the *n*th element are sorted. It thus makes fewer guarantees than `partial_sort`, which orders the elements in the range below some chosen element, and may be used as a faster alternative to `partial_sort` when the ordering of the lower range is not required.

Elements are equivalent, but not necessarily equal, if neither is less than the other.

The average of a sort complexity is linear with respect to $* \text{last} - \text{first} *$.

Example

```

// alg_nth_elem.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 ) {
    return elem1 > elem2;
}

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1.push_back( 3 * i );

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
        v1.push_back( 3 * ii + 1 );

    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
        v1.push_back( 3 * iii + 2 );

    cout << "Original vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    nth_element(v1.begin( ), v1.begin( ) + 3, v1.end( ) );
    cout << "Position 3 partitioned vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order, specify binary predicate
    nth_element( v1.begin( ), v1.begin( ) + 4, v1.end( ),
        greater<int>( ) );
    cout << "Position 4 partitioned (greater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    random_shuffle( v1.begin( ), v1.end( ) );
    cout << "Shuffled vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    nth_element( v1.begin( ), v1.begin( ) + 5, v1.end( ), UDgreater );
    cout << "Position 5 partitioned (UDgreater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

none_of

Returns **true** when a condition is never present among elements in the given range.

```
template<class InputIterator, class BinaryPredicate>
bool none_of(InputIterator first, InputIterator last, BinaryPredicate comp);
```

Parameters

first

An input iterator that indicates where to start to check a range of elements for a condition.

last

An input iterator that indicates the end of a range of elements.

comp

The condition to test for. This is provided by a user-defined predicate function object that defines the condition. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns **true** if the condition is not detected at least once in the indicated range, and **false** if the condition is detected.

Remarks

The template function returns **true** only if, for some `N` in the range `[0, last - first)`, the predicate `comp(*(first + N))` is always **false**.

partial_sort

Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
void partial_sort(
    RandomAccessIterator first,
    RandomAccessIterator sortEnd,
    RandomAccessIterator last);

template<class RandomAccessIterator, class BinaryPredicate>
void partial_sort(
    RandomAccessIterator first,
    RandomAccessIterator sortEnd,
    RandomAccessIterator last,
    BinaryPredicate comp);
```

Parameters

first

A random-access iterator addressing the position of the first element in the range to be sorted.

sortEnd

A random-access iterator addressing the position one past the final element in the subrange to be sorted.

last

A random-access iterator addressing the position one past the final element in the range to be partially sorted.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_sort` does preserve this original ordering.

The average partial sort complexity is $O((\text{last} - \text{first}) \log (\text{sortEnd} - \text{first}))$.

Example

```

// alg_partial_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector <int> v1;
    vector <int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
    {
        v1.push_back( 2 * ii + 1 );
    }

    cout << "Original vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    partial_sort(v1.begin( ), v1.begin( ) + 6, v1.end( ) );
    cout << "Partially sorted vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To partially sort in descending order, specify binary predicate
    partial_sort(v1.begin( ), v1.begin( ) + 4, v1.end( ), greater<int>( ) );
    cout << "Partially resorted (greater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    partial_sort(v1.begin( ), v1.begin( ) + 8, v1.end( ),
UDgreater );
    cout << "Partially resorted (UDgreater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

```
Original vector:
v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Partially sorted vector:
v1 = ( 0 1 2 3 4 5 10 8 6 7 9 11 )
Partially resorted (greater) vector:
v1 = ( 11 10 9 8 0 1 2 3 4 5 6 7 )
Partially resorted (UDgreater) vector:
v1 = ( 11 10 9 8 7 6 5 4 0 1 2 3 )
```

partial_sort_copy

Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator first1,
    InputIterator last1,
    RandomAccessIterator first2,
    RandomAccessIterator last2 );

template<class InputIterator, class RandomAccessIterator, class BinaryPredicate>
RandomAccessIterator partial_sort_copy(
    InputIterator first1,
    InputIterator last1,
    RandomAccessIterator first2,
    RandomAccessIterator last2,
    BinaryPredicate comp);
```

Parameters

first1

An input iterator addressing the position of the first element in the source range.

last1

An input iterator addressing the position one past the final element in the source range.

first2

A random-access iterator addressing the position of the first element in the sorted destination range.

last2

A random-access iterator addressing the position one past the final element in the sorted destination range.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A random-access iterator addressing the element in the destination range one position beyond the last element inserted from the source range.

Remarks

The source and destination ranges must not overlap and must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The binary predicate must provide a strict weak ordering so that elements that are not equivalent are ordered, but elements that are equivalent are not. Two elements are equivalent under less than, but not necessarily equal, if neither is less than the other.

Example

```
// alg_partial_sort_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    list<int> list1;
    vector<int>::iterator iter1, iter2;
    list<int>::iterator list1_Iter, list1_inIter;

    int i;
    for (i = 0; i <= 9; i++)
        v1.push_back(i);

    random_shuffle(v1.begin(), v1.end());

    list1.push_back(60);
    list1.push_back(50);
    list1.push_back(20);
    list1.push_back(30);
    list1.push_back(40);
    list1.push_back(10);

    cout << "Vector v1 = ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;

    cout << "List list1 = ( " ;
    for (list1_Iter = list1.begin();
        list1_Iter != list1.end();
        list1_Iter++)
        cout << *list1_Iter << " ";
    cout << ")" << endl;

    // Copying a partially sorted copy of list1 into v1
    vector<int>::iterator result1;
    result1 = partial_sort_copy(list1.begin(), list1.end(),
        v1.begin(), v1.begin() + 3);

    cout << "List list1 Vector v1 = ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;
    cout << "The first v1 element one position beyond"
        << "\n the last 1 element inserted was " << *result1
        << "." << endl;

    // Copying a partially sorted copy of list1 into v2
    int ii;
    for (ii = 0; ii <= 9; ii++)
        v2.push_back(ii);

    random_shuffle(v2.begin(), v2.end());
    vector<int>::iterator result2;
    result2 = partial_sort_copy(list1.begin(), list1.end(),
        v2.begin(), v2.begin() + 6);

    cout << "List list1 into Vector v2 = ( " ;
    for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
        cout << *iter2 << " ";
```

```

    cout << ")" << endl;
    cout << "The first v2 element one position beyond"
        << "\n the last L 1 element inserted was " << *result2
        << "." << endl;
}

```

partition

Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.

```

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Predicate comp);

```

Parameters

first

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

comp

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A predicate takes a single argument and returns **true** or **false**.

Return Value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements a and b are equivalent, but not necessarily equal, if both $Pr(a, b)$ is false and $Pr(b, a)$ if false, where Pr is the parameter-specified predicate. The `partition` algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_partition` does preserve this original ordering.

The complexity is linear: there are $(last - first)$ applications of *comp* and at most $(last - first)/2$ swaps.

Example

```

// alg_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value ) {
    return value > 5;
}

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 10 ; i++ )
    {
        v1.push_back( i );
    }
    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Partition the range with predicate greater10
    partition ( v1.begin( ), v1.end( ), greater5 );
    cout << "The partitioned set of elements in v1 is: ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

partition_copy

Copies elements for which a condition is **true** to one destination, and for which the condition is **false** to another. The elements must come from a specified range.

```

template<class InputIterator, class OutputIterator1, class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2>
partition_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator1 dest1,
    OutputIterator2 dest2,
    Predicate pred);

```

Parameters

first

An input iterator that indicates the beginning of a range to check for a condition.

last

An input iterator that indicates the end of a range.

dest1

An output iterator used to copy elements that return true for a condition tested by using *_Pred*.

dest2

An output iterator used to copy elements that return false for a condition tested by using *_Pred*.

_Pred

The condition to test for. This is provided by a user-defined predicate function object that defines the condition to be tested. A predicate takes a single argument and returns **true** or **false**.

Remarks

The template function copies each element `x` in `[first,last)` to `*dest1++` if `_Pred(x)` is true, or to `*dest2++` if not. It returns `pair<OutputIterator1, OutputIterator2>(dest1, dest2)`.

partition_point

Returns the first element in the given range that does not satisfy the condition. The elements are sorted so that those that satisfy the condition come before those that do not.

```
template<class ForwardIterator, class Predicate>
ForwardIterator partition_point(
    ForwardIterator first,
    ForwardIterator last,
    Predicate comp);
```

Parameters

first

A `ForwardIterator` that indicates the start of a range to check for a condition.

last

A `ForwardIterator` that indicates the end of a range.

comp

The condition to test for. This is provided by a user-defined predicate function object that defines the condition to be satisfied by the element being searched for. A predicate takes a single argument and returns **true** or **false**.

Return Value

Returns a `ForwardIterator` that refers to the first element that does not fulfill the condition tested for by *comp*, or returns *last* if one is not found.

Remarks

The template function finds the first iterator `it` in `[first, last)` for which `comp(*it)` is **false**. The sequence must be ordered by *comp*.

pop_heap

Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.

```
template<class RandomAccessIterator>
void pop_heap( RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class BinaryPredicate>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
```

Parameters

first

A random-access iterator addressing the position of the first element in the heap.

last

A random-access iterator addressing the position one past the final element in the heap.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The `pop_heap` algorithm is the inverse of the operation performed by the `push_heap` algorithm, in which an element at the next-to-last position of a range is added to a heap consisting of the prior elements in the range, in the case when the element being added to the heap is larger than any of the elements already in the heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the C++ Standard Library container adaptor [priority_queue Class](#).

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most $\log(*last - first)$ comparisons.

Example


```

// alg_pop_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 1 ; i <= 9 ; i++ )
        v1.push_back( i );

    // Make v1 a heap with default less than ordering
    random_shuffle( v1.begin( ), v1.end( ) );
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Add an element to the back of the heap
    v1.push_back( 10 );
    push_heap( v1.begin( ), v1.end( ) );
    cout << "The reheaped v1 with 10 added is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove the largest element from the heap
    pop_heap( v1.begin( ), v1.end( ) );
    cout << "The heap v1 with 10 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl << endl;

    // Make v1 a heap with greater-than ordering with a 0 element
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    v1.push_back( 0 );
    push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The 'greater than' reheaped v1 puts the smallest "
        << "element first:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Application of pop_heap to remove the smallest element
    pop_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The 'greater than' heaped v1 with the smallest element\n "
        << "removed from the heap is: ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

prev_permutation

Reorders the elements in a range so that the original ordering is replaced by the lexicographically previous greater permutation if it exists, where the sense of previous may be specified with a binary predicate.

```

template<class BidirectionalIterator>
bool prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last);

template<class BidirectionalIterator, class BinaryPredicate>
bool prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last,
    BinaryPredicate comp);

```

Parameters

first

A bidirectional iterator pointing to the position of the first element in the range to be permuted.

last

A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

true if the lexicographically previous permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically largest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less-than comparable to ensure that the previous permutation is well defined.

The complexity is linear, with at most $(\text{last} - \text{first})/2$ swaps.

Example

```

// alg_prev_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<< ( ostream& osIn, const CInt& rhs );

class CInt {
public:
    CInt( int n = 0 ) : m_nVal( n ){ }
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){ }
    CInt& operator=( const CInt& rhs ) { m_nVal =
        rhs.m_nVal; return *this; }
    bool operator< ( const CInt& rhs ) const
        { return ( m_nVal < rhs.m_nVal ); }
    friend ostream& operator<< ( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;

```

```

    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs ) {
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 ) {
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main() {
    // Reordering the elements of type CInt in a deque
    // using the prev_permutation algorithm
    CInt c1 = 1, c2 = 5, c3 = 10;
    bool deq1Result;
    deque<CInt> deq1, deq2, deq3;
    deque<CInt>::iterator d1_Iter;

    deq1.push_back ( c1 );
    deq1.push_back ( c2 );
    deq1.push_back ( c3 );

    cout << "The original deque of CInts is deq1 = (" ;
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    deq1Result = prev_permutation ( deq1.begin( ), deq1.end( ) );

    if ( deq1Result )
        cout << "The lexicographically previous permutation "
            << "exists and has \nreplaced the original "
            << "ordering of the sequence in deq1." << endl;
    else
        cout << "The lexicographically previous permutation doesn't "
            << "exist\n and the lexicographically "
            << "smallest permutation\n has replaced the "
            << "original ordering of the sequence in deq1." << endl;

    cout << "After one application of prev_permutation,\n deq1 = (" ;
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl << endl;

    // Permutating vector elements with binary function mod_lesser
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = -3 ; i <= 3 ; i++ )
        v1.push_back( i );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ); Iter1 != v1.end( ); Iter1++ )
        cout << *Iter1 << " ";
    cout << " )." << endl;

    prev_permutation ( v1.begin( ), v1.end( ), mod_lesser );

    cout << "After the first prev_permutation, vector v1 is:\n v1 = ( " ;

```

```

for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

int iii = 1;
while ( iii <= 5 ) {
    prev_permutation ( v1.begin( ), v1.end( ), mod_lesser );
    cout << "After another prev_permutation of vector v1,\n v1 =  ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ;Iter1 ++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
    iii++;
}
}

```

The original deque of CInts is `deq1 = (CInt(1), CInt(5), CInt(10))`.
The lexicographically previous permutation doesn't exist
and the lexicographically smallest permutation
has replaced the original ordering of the sequence in `deq1`.
After one application of `prev_permutation`,
`deq1 = (CInt(10), CInt(5), CInt(1))`.

Vector `v1` is (-3 -2 -1 0 1 2 3).
After the first `prev_permutation`, vector `v1` is:
`v1 = (-3 -2 0 3 2 1 -1)`.
After another `prev_permutation` of vector `v1`,
`v1 = (-3 -2 0 3 -1 2 1)`.
After another `prev_permutation` of vector `v1`,
`v1 = (-3 -2 0 3 -1 1 2)`.
After another `prev_permutation` of vector `v1`,
`v1 = (-3 -2 0 2 3 1 -1)`.
After another `prev_permutation` of vector `v1`,
`v1 = (-3 -2 0 2 -1 3 1)`.
After another `prev_permutation` of vector `v1`,
`v1 = (-3 -2 0 2 -1 1 3)`.

push_heap

Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.

```

template<class RandomAccessIterator>
void push_heap( RandomAccessIterator first, RandomAccessIterator last );

template<class RandomAccessIterator, class BinaryPredicate>
void push_heap( RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);

```

Parameters

first

A random-access iterator addressing the position of the first element in the heap.

last

A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The element must first be pushed back to the end of an existing heap and then the algorithm is used to add this

element to the existing heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the C++ Standard Library container adaptor [priority_queue Class](#).

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most $\log (last - first)$ comparisons.

Example

```

// alg_push_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 1 ; i <= 9 ; i++ )
        v1.push_back( i );

    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with default less than ordering
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Add an element to the heap
    v1.push_back( 10 );
    cout << "The heap v1 with 10 pushed back is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    push_heap( v1.begin( ), v1.end( ) );
    cout << "The reheaped v1 with 10 added is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl << endl;

    // Make v1 a heap with greater than ordering
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater-than heaped version of v1 is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    v1.push_back(0);
    cout << "The greater-than heap v1 with 11 pushed back is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater than reheaped v1 with 11 added is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

random_shuffle

The `std::random_shuffle()` function is deprecated, replaced by [std::shuffle](#). For a code example and more information, see [<random>](#) and the Stack Overflow post [Why are std::random_shuffle methods being deprecated in C++14?](#).

remove

Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Type>
ForwardIterator remove(ForwardIterator first, ForwardIterator last, const Type& val);
```

Parameters

first

A forward iterator addressing the position of the first element in the range from which elements are being removed.

last

A forward iterator addressing the position one past the final element in the range from which elements are being removed.

val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are $(\text{last} - \text{first})$ comparisons for equality.

The [list class](#) has a more efficient member function version of `remove`, which also relinks pointers.

Example

```

// alg_remove.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value of 7
    new_end = remove ( v1.begin( ), v1.end( ), 7 );

    cout << "Vector v1 with value 7 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To change the sequence size, use erase
    v1.erase (new_end, v1.end( ) );

    cout << "Vector v1 resized with value 7 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

remove_copy

Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```

template<class InputIterator, class OutputIterator, class Type>
OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, const Type& val);

```

Parameters

first

An input iterator addressing the position of the first element in the range from which elements are being removed.

last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

val

The value that is to be removed from the range.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are $(\text{last} - \text{first})$ comparisons for equality and at most $(\text{last} - \text{first})$ assignments.

Example

```
// alg_remove_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2(10);
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle (v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:    ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value of 7
    new_end = remove_copy ( v1.begin( ), v1.end( ), v2.begin( ), 7 );

    cout << "Vector v1 is left unchanged as ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is a copy of v1 with the value 7 removed:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;
}
```

remove_copy_if

Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);
```

Parameters

first

An input iterator addressing the position of the first element in the range from which elements are being removed.

last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are $(last - first)$ comparisons for equality and at most $(last - first)$ assignments.

For information on how these functions behave, see [Checked Iterators](#).

Example

```

// alg_remove_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
    return value > 6;
}

int main() {
    using namespace std;
    vector<int> v1, v2(10);
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:      ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value greater than 6
    new_end = remove_copy_if ( v1.begin( ), v1.end( ),
        v2.begin( ), greater6 );

    cout << "After the appliation of remove_copy_if to v1,\n "
        << "vector v1 is left unchanged as ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is a copy of v1 with values greater "
        << "than 6 removed:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != new_end ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;
}

```

remove_if

Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```

template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(
    ForwardIterator first,
    ForwardIterator last,
    Predicate pred);

```

Parameters

first

A forward iterator pointing to the position of the first element in the range from which elements are being removed.

last

A forward iterator pointing to the position one past the final element in the range from which elements are being removed.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return Value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are $(\text{last} - \text{first})$ comparisons for equality.

List has a more efficient member function version of remove which relinks pointers.

Example

```

// alg_remove_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
    return value > 6;
}

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements satisfying predicate greater6
    new_end = remove_if (v1.begin( ), v1.end( ), greater6 );

    cout << "Vector v1 with elements satisfying greater6 removed is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To change the sequence size, use erase
    v1.erase (new_end, v1.end( ) );

    cout << "Vector v1 resized elements satisfying greater6 removed is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

replace

Examines each element in a range and replaces it if it matches a specified value.

```

template<class ForwardIterator, class Type>
void replace(
    ForwardIterator first,
    ForwardIterator last,
    const Type& oldVal,
    const Type& newVal);

```

Parameters

first

A forward iterator pointing to the position of the first element in the range from which elements are being replaced.

last

A forward iterator pointing to the position one past the final element in the range from which elements are being replaced.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are $(\text{last} - \text{first})$ comparisons for equality and at most $(\text{last} - \text{first})$ assignments of new values.

Example

```
// alg_replace.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle (v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements with a value of 7 with a value of 700
    replace (v1.begin( ), v1.end( ), 7 , 700);

    cout << "The vector v1 with a value 700 replacing that of 7 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

replace_copy

Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.

```
template<class InputIterator, class OutputIterator, class Type>
OutputIterator replace_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    const Type& oldVal,
    const Type& newVal);
```

Parameters

first

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

result

An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

_OldVal

The old value of the elements being replaced.

_NewVal

The new value being assigned to the elements with the old value.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are $(\text{last} - \text{first})$ comparisons for equality and at most $(\text{last} - \text{first})$ assignments of new values.

Example

```

// alg_replace_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    list<int> L1 (15);
    vector<int>::iterator Iter1;
    list<int>::iterator L_Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );

    int iii;
    for ( iii = 0 ; iii <= 15 ; iii++ )
        v1.push_back( 1 );

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements in one part of a vector with a value of 7
    // with a value of 70 and copy into another part of the vector
    replace_copy ( v1.begin( ), v1.begin( ) + 14, v1.end( ) - 15, 7 , 70);

    cout << "The vector v1 with a value 70 replacing that of 7 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements in a vector with a value of 70
    // with a value of 1 and copy into a list
    replace_copy ( v1.begin( ), v1.begin( ) + 14, L1.begin( ), 7 , 1);

    cout << "The list copy L1 of v1 with the value 0 replacing "
        << "that of 7 is:\n ( " ;
    for ( L_Iter1 = L1.begin( ) ; L_Iter1 != L1.end( ) ; L_Iter1++ )
        cout << *L_Iter1 << " ";
    cout << ")." << endl;
}

```

replace_copy_if

Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.


```
template<class InputIterator, class OutputIterator, class Predicate, class Type>
OutputIterator replace_copy_if(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    Predicate pred,
    const Type& val);
```

Parameters

first

An input iterator pointing to the position of the first element in the range from which elements are being replaced.

last

An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

val

The new value being assigned to the elements whose old value satisfies the predicate.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear; there are $(\text{last} - \text{first})$ comparisons for equality and at most $(\text{last} - \text{first})$ assignments of new values.

Example

```

// alg_replace_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
    return value > 6;
}

int main() {
    using namespace std;
    vector <int> v1;
    list <int> L1 (13);
    vector <int>::iterator Iter1;
    list <int>::iterator L_Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );

    int iii;
    for ( iii = 0 ; iii <= 13 ; iii++ )
        v1.push_back( 1 );

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements with a value of 7 in the 1st half of a vector
    // with a value of 70 and copy it into the 2nd half of the vector
    replace_copy_if ( v1.begin( ), v1.begin( ) + 14, v1.end( ) - 14,
        greater6 , 70);

    cout << "The vector v1 with values of 70 replacing those greater"
        << "\n than 6 in the 1st half & copied into the 2nd half is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements in a vector with a value of 70
    // with a value of 1 and copy into a list
    replace_copy_if ( v1.begin( ), v1.begin( ) + 13, L1.begin( ),
        greater6 , -1 );

    cout << "A list copy of vector v1 with the value -1\n replacing "
        << "those greater than 6 is:\n ( " ;
    for ( L_Iter1 = L1.begin( ) ; L_Iter1 != L1.end( ) ; L_Iter1++ )
        cout << *L_Iter1 << " ";
    cout << ")." << endl;
}

```

replace_if

Examines each element in a range and replaces it if it satisfies a specified predicate.

```
template<class ForwardIterator, class Predicate, class Type>
void replace_if(
    ForwardIterator first,
    ForwardIterator last,
    Predicate pred,
    const Type& val);
```

Parameters

first

A forward iterator pointing to the position of the first element in the range from which elements are being replaced.

last

An iterator pointing to the position one past the final element in the range from which elements are being replaced.

_Pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

val

The new value being assigned to the elements whose old value satisfies the predicate.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The algorithm `replace_if` is a generalization of the algorithm `replace`, allowing any predicate to be specified, rather than equality to a specified constant value.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear: there are $(\text{last} - \text{first})$ comparisons for equality and at most $(\text{last} - \text{first})$ assignments of new values.

Example

```

// alg_replace_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value ) {
    return value > 6;
}

int main() {
    using namespace std;
    vector <int> v1;
    vector <int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements satisfying the predicate greater6
    // with a value of 70
    replace_if ( v1.begin( ), v1.end( ), greater6 , 70);

    cout << "The vector v1 with a value 70 replacing those\n "
        << "elements satisfying the greater6 predicate is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

reverse

Reverses the order of the elements within a range.

```

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

```

Parameters

first

A bidirectional iterator pointing to the position of the first element in the range within which the elements are being permuted.

last

A bidirectional iterator pointing to the position one past the final element in the range within which the elements are being permuted.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Example

```

// alg_reverse.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Reverse the elements in the vector
    reverse (v1.begin( ), v1.end( ) );

    cout << "The modified vector v1 with values reversed is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

```

```

The original vector v1 is:
( 0 1 2 3 4 5 6 7 8 9 ).
The modified vector v1 with values reversed is:
( 9 8 7 6 5 4 3 2 1 0 ).

```

reverse_copy

Reverses the order of the elements within a source range while copying them into a destination range

```

template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(
    BidirectionalIterator first,
    BidirectionalIterator last,
    OutputIterator result);

```

Parameters

first

A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.

last

A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.

result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

Return Value

An output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Example

```
// alg_reverse_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2( 10 );
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Reverse the elements in the vector
    reverse_copy( v1.begin( ), v1.end( ), v2.begin( ) );

    cout << "The copy v2 of the reversed vector v1 is:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    cout << "The original vector v1 remains unmodified as:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

rotate

Exchanges the elements in two adjacent ranges.

```
template<class ForwardIterator>
void rotate(
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be rotated.

middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

Last

A forward iterator addressing the position one past the final element in the range to be rotated.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear with at most $(\text{last} - \text{first})$ swaps.

Example

```

// alg_rotate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    deque<int> d1;
    vector<int>::iterator v1Iter1;
    deque<int>::iterator d1Iter1;

    int i;
    for ( i = -3 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii =0 ; ii <= 5 ; ii++ )
    {
        d1.push_back( ii );
    }

    cout << "Vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    rotate ( v1.begin( ), v1.begin( ) + 3 , v1.end( ) );
    cout << "After rotating, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= d1.end( ) - d1.begin( ) ) {
        rotate ( d1.begin( ), d1.begin( ) + 1 , d1.end( ) );
        cout << "After the rotation of a single deque element to the back,\n d1 is ( " ;
        for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
            cout << *d1Iter1 << " ";
        cout << ")." << endl;
        iii++;
    }
}

```


Vector v1 is (-3 -2 -1 0 1 2 3 4 5).
After rotating, vector v1 is (0 1 2 3 4 5 -3 -2 -1).
The original deque d1 is (0 1 2 3 4 5).
After the rotation of a single deque element to the back,
d1 is (1 2 3 4 5 0).
After the rotation of a single deque element to the back,
d1 is (2 3 4 5 0 1).
After the rotation of a single deque element to the back,
d1 is (3 4 5 0 1 2).
After the rotation of a single deque element to the back,
d1 is (4 5 0 1 2 3).
After the rotation of a single deque element to the back,
d1 is (5 0 1 2 3 4).
After the rotation of a single deque element to the back,
d1 is (0 1 2 3 4 5).

rotate_copy

Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last,
    OutputIterator result );
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be rotated.

middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.

_Last A forward iterator addressing the position one past the final element in the range to be rotated.

result

An output iterator addressing the position of the first element in the destination range.

Return Value

An output iterator addressing the position one past the final element in the destination range.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear with at most $(\text{last} - \text{first})$ swaps.

Example

```

// alg_rotate_copy.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1 , v2 ( 9 );
    deque<int> d1 , d2 ( 6 );
    vector<int>::iterator v1Iter , v2Iter;
    deque<int>::iterator d1Iter , d2Iter;

    int i;
    for ( i = -3 ; i <= 5 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii =0 ; ii <= 5 ; ii++ )
        d1.push_back( ii );

    cout << "Vector v1 is ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    rotate_copy ( v1.begin( ) , v1.begin( ) + 3 , v1.end( ) , v2.begin( ) );
    cout << "After rotating, the vector v1 remains unchanged as:\n v1 = ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    cout << "After rotating, the copy of vector v1 in v2 is:\n v2 = ( " ;
    for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ;v2Iter ++ )
        cout << *v2Iter << " ";
    cout << ")." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter = d1.begin( ) ; d1Iter != d1.end( ) ;d1Iter ++ )
        cout << *d1Iter << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= d1.end( ) - d1.begin( ) )
    {
        rotate_copy ( d1.begin( ) , d1.begin( ) + iii , d1.end( ) , d2.begin( ) );
        cout << "After the rotation of a single deque element to the back,\n d2 is ( " ;
        for ( d2Iter = d2.begin( ) ; d2Iter != d2.end( ) ;d2Iter ++ )
            cout << *d2Iter << " ";
        cout << ")." << endl;
        iii++;
    }
}

```

search

Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.

```

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class Predicate>
ForwardIterator1 search(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2
    Predicate comp);

```

Parameters

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the final element in the range to be searched.

first2

A forward iterator addressing the position of the first element in the range to be matched.

last2

A forward iterator addressing the position one past the final element in the range to be matched.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

Average complexity is linear with respect to the size of the searched range, and worst case complexity is also linear with respect to the size of the sequence being searched for.

Example

```

// alg_search.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice (int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

```

```

int main() {
    using namespace std;
    vector<int> v1, v2;
    list<int> L1;
    vector<int>::iterator Iter1, Iter2;
    list<int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
    for ( ii = 4 ; ii <= 5 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Searching v1 for first match to L1 under identity
    vector<int>::iterator result1;
    result1 = search (v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

    if ( result1 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is at least one match of L1 in v1"
            << "\n and the first one begins at "
            << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    result2 = search (v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice );

    if ( result2 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a sequence of elements in v1 that "
            << "are equivalent\n to those in v2 under the binary "
            << "predicate twice\n and the first one begins at position "
            << result2 - v1.begin( ) << "." << endl;
}

```

```

Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 20 25 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
and the first one begins at position 4.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.

```

search_n

Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.

```

template<class ForwardIterator1, class Diff2, class Type>
ForwardIterator1 search_n(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    Diff2 count,
    const Type& val);

template<class ForwardIterator1, class Diff2, class Type, class BinaryPredicate>
ForwardIterator1 search_n(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    Diff2 count,
    const Type& val,
    BinaryPredicate comp);

```

Parameters

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the final element in the range to be searched.

count

The size of the subsequence being searched for.

val

The value of the elements in the sequence being searched for.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Complexity is linear with respect to the size of the searched.

Example

```
// alg_search_n.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is 1/2 of the first
bool one_half ( int elem1, int elem2 )
{
    return elem1 == 2 * elem2;
}

int main()
{
    using namespace std;
    vector <int> v1, v2;
    vector <int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( 5 );
    }

    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( 10 );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Searching v1 for first match to (5 5 5) under identity
    vector <int>::iterator result1;
    result1 = search_n ( v1.begin( ), v1.end( ), 3, 5 );

    if ( result1 == v1.end( ) )
        cout << "There is no match for a sequence ( 5 5 5 ) in v1."
              << endl;
    else
        cout << "There is at least one match of a sequence ( 5 5 5 )"
              << "\n in v1 and the first one begins at "
              << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for first match to (5 5 5) under one_half
    vector <int>::iterator result2;
    result2 = search_n (v1.begin( ), v1.end( ), 3, 5, one_half );

    if ( result2 == v1.end( ) )
        cout << "There is no match for a sequence ( 5 5 5 ) in v1"
              << " under the equivalence predicate one_half." << endl;
```

```

else
    cout << "There is a match of a sequence ( 5 5 5 ) "
        << "under the equivalence\n predicate one_half "
        << "in v1 and the first one begins at "
        << "position "<< result2 - v1.begin( ) << "." << endl;
}

```

```

Vector v1 = ( 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 10 10 10 )
There is at least one match of a sequence ( 5 5 5 )
in v1 and the first one begins at position 6.
There is a match of a sequence ( 5 5 5 ) under the equivalence
predicate one_half in v1 and the first one begins at position 15.

```

set_difference

Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate>
OutputIterator set_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The

binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the first source range.

The sorted source ranges must each be arranged as a precondition to the application of the `set_difference` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm merge.

The value types of the input iterators need be less-than-comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element such that there are more in the first source range than in the second, then the destination range will contain the number by which the occurrences of those elements in the first source range exceed the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most $2 * ((last1 - first1) - (last2 - first2)) - 1$ comparisons for nonempty source ranges.

Example

```
// alg_set_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector <int> v1a, v1b, v1 ( 12 );
    vector <int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -1 ; i <= 4 ; i++ )
    {
        v1a.push_back( i );
    }
}
```



```

int ii;
for ( ii = -3 ; ii <= 0 ; ii++ )
{
    v1b.push_back( ii );
}

cout << "Original vector v1a with range sorted by the\n "
    << "binary predicate less than is v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
    << "binary predicate less than is v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin( ) , v2a.end( ) , greater<int>( ) );
sort ( v2b.begin( ) , v2b.end( ) , greater<int>( ) );

cout << "Original vector v2a with range sorted by the\n "
    << "binary predicate greater is v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
    << "binary predicate greater is v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector<int> v3a ( v1a ) , v3b ( v1b ) , v3 ( v1 );
vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin( ) , v3a.end( ) , mod_lesser );
sort ( v3b.begin( ) , v3b.end( ) , mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
    << "binary predicate mod_lesser is v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
    << "binary predicate mod_lesser is v3b = ( " ;
for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
    cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a difference in ascending
// order with the default binary predicate less<int>( )
Result1 = set_difference ( v1a.begin( ) , v1a.end( ) ,
    v1b.begin( ) , v1b.end( ) , v1.begin( ) );
cout << "Set_difference of source ranges with default order,"
    << "\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a difference in descending
// order specify binary predicate greater<int>( )
Result2 = set_difference ( v2a.begin( ) , v2a.end( ) ,
    v2b.begin( ) , v2b.end( ) , v2.begin( ) , greater<int>( ) );

```

```

cout << "Set_difference of source ranges with binary"
    << "predicate greater specified,\n vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")." << endl;

// To combine into a difference applying a user
// defined binary predicate mod_lesser
Result3 = set_difference ( v3a.begin( ), v3a.end( ),
    v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
cout << "Set_difference of source ranges with binary "
    << "predicate mod_lesser specified,\n vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")." << endl;
}

```

set_intersection

Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate>
OutputIterator set_intersection(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

_Result An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the intersection of the two source ranges.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the merge algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most $2 * ((last1 - first1) + (last2 - first2)) - 1$ comparisons for nonempty source ranges.

Example

```
// alg_set_intersection.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>    // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 ) {
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main() {
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less than ordering
    int i;
    for ( i = -1 ; i <= 3 ; i++ )
        v1a.push_back( i );

    int ii;
    for ( ii = -3 ; ii <= 1 ; ii++ )
        v1b.push_back( ii );
```

```

cout << "Original vector v1a with range sorted by the\n "
    << "binary predicate less than is v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
    << "binary predicate less than is v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector <int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin( ) , v2a.end( ) , greater<int>( ) );
sort ( v2b.begin( ) , v2b.end( ) , greater<int>( ) );

cout << "Original vector v2a with range sorted by the\n "
    << "binary predicate greater is v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
    << "binary predicate greater is v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector <int> v3a ( v1a ) , v3b ( v1b ) , v3 ( v1 );
vector <int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin( ) , v3a.end( ) , mod_lesser );
sort ( v3b.begin( ) , v3b.end( ) , mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
    << "binary predicate mod_lesser is v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
    << "binary predicate mod_lesser is v3b = ( " ;
for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
    cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into an intersection in ascending order with the
// default binary predicate less <int>( )
Result1 = set_intersection ( v1a.begin( ) , v1a.end( ) ,
    v1b.begin( ) , v1b.end( ) , v1.begin( ) );
cout << "Intersection of source ranges with default order,"
    << "\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; ++Iter1 )
    cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into an intersection in descending order, specify
// binary predicate greater<int>( )
Result2 = set_intersection ( v2a.begin( ) , v2a.end( ) ,
    v2b.begin( ) , v2b.end( ) , v2.begin( ) , greater <int>( ) );
cout << "Intersection of source ranges with binary predicate"
    << " greater specified,\n vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; ++Iter2 )
    cout << *Iter2 << " ";
cout << ")." << endl;

```

```

// To combine into an intersection applying a user-defined
// binary predicate mod_lesser
Result3 = set_intersection ( v3a.begin( ), v3a.end( ),
    v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
cout << "Intersection of source ranges with binary predicate "
    << "mod_lesser specified,\n vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; ++Iter3 )
    cout << *Iter3 << " ";
cout << ")." << endl;
}

```

set_symmetric_difference

Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate>
OutputIterator set_symmetric_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

_Result An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the symmetric difference of the two source ranges.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge*` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm `merge`.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the absolute value of the number by which the occurrences of those elements in the one of the source ranges exceeds the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most $2 * ((last1 - first1) - (last2 - first2)) - 1$ comparisons for nonempty source ranges.

Example

```
// alg_set_sym_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -1 ; i <= 4 ; i++ )
    {
        v1a.push_back( i );
    }

    int ii;
    for ( ii = -3 ; ii <= 0 ; ii++ )
    {
```

```

    v1b.push_back( ii );
}

cout << "Original vector v1a with range sorted by the\n "
    << "binary predicate less than is v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
    << "binary predicate less than is v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin( ) , v2a.end( ) , greater<int>( ) );
sort ( v2b.begin( ) , v2b.end( ) , greater<int>( ) );

cout << "Original vector v2a with range sorted by the\n "
    << "binary predicate greater is v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
    << "binary predicate greater is v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector<int> v3a ( v1a ) , v3b ( v1b ) , v3 ( v1 );
vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin( ) , v3a.end( ) , mod_lesser );
sort ( v3b.begin( ) , v3b.end( ) , mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
    << "binary predicate mod_lesser is v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
    << "binary predicate mod_lesser is v3b = ( " ;
for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
    cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a symmetric difference in ascending
// order with the default binary predicate less<int>( )
Result1 = set_symmetric_difference ( v1a.begin( ) , v1a.end( ) ,
    v1b.begin( ) , v1b.end( ) , v1.begin( ) );
cout << "Set_symmetric_difference of source ranges with default order,"
    << "\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a symmetric difference in descending
// order, specify binary predicate greater<int>( )
Result2 = set_symmetric_difference ( v2a.begin( ) , v2a.end( ) ,
    v2b.begin( ) , v2b.end( ) , v2.begin( ) , greater<int>( ) );
cout << "Set_symmetric_difference of source ranges with binary"
    << "predicate greater specified,\n vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
    cout << *Iter2 << " ";

```

```

        cout << *Iter2 << " ",
        cout << ")." << endl;

// To combine into a symmetric difference applying a user
// defined binary predicate mod_lesser
Result3 = set_symmetric_difference ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
cout << "Set_symmetric_difference of source ranges with binary "
        << "predicate mod_lesser specified,\n vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")." << endl;
}

```

set_union

Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate>
OutputIterator set_union(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    BinaryPredicate comp );

```

Parameters

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

_ Result An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the union of the two source ranges.

comp

User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return Value

An output iterator addressing the position one past the last element in the sorted destination range representing the union of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm `merge`.

The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most $2 * ((last1 - first1) - (last2 - first2)) - 1$ comparisons.

Example

```
// alg_set_union.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector <int> v1a, v1b, v1 ( 12 );
    vector <int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less than ordering
    int i;
    for ( i = -1 ; i <= 3 ; i++ )
    {
        v1a.push_back( i );
    }

    int ii;
    for ( ii = -3 ; ii <= 1 ; ii++ )
    {
        v1b.push_back( ii );
    }
}
```

```

cout << "Original vector v1a with range sorted by the\n "
    << "binary predicate less than is  v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
    << "binary predicate less than is  v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector <int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin( ) , v2a.end( ) , greater<int>( ) );
sort ( v2b.begin( ) , v2b.end( ) , greater<int>( ) );

cout << "Original vector v2a with range sorted by the\n "
    << "binary predicate greater is  v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
    << "binary predicate greater is  v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector <int> v3a ( v1a ) , v3b ( v1b ) , v3 ( v1 );
vector <int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin( ) , v3a.end( ) , mod_lesser );
sort ( v3b.begin( ) , v3b.end( ) , mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
    << "binary predicate mod_lesser is  v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
    << "binary predicate mod_lesser is  v3b = ( " ;
for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
    cout << *Iter3b << " ";
cout << ")." << endl;

// To combine into a union in ascending order with the default
// binary predicate less <int>( )
Result1 = set_union ( v1a.begin( ) , v1a.end( ) ,
    v1b.begin( ) , v1b.end( ) , v1.begin( ) );
cout << "Union of source ranges with default order,"
    << "\n vector v1mod = ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

// To combine into a union in descending order, specify binary
// predicate greater<int>( )
Result2 = set_union ( v2a.begin( ) , v2a.end( ) ,
    v2b.begin( ) , v2b.end( ) , v2.begin( ) , greater <int>( ) );
cout << "Union of source ranges with binary predicate greater "
    << "specified,\n vector v2mod = ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")." << endl;

```

```
// To combine into a union applying a user-defined
// binary predicate mod_lesser
Result3 = set_union ( v3a.begin( ), v3a.end( ),
    v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
cout << "Union of source ranges with binary predicate "
    << "mod_lesser specified,\n vector v3mod = ( " ; ;
for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
    cout << *Iter3 << " ";
cout << ")." << endl;
}
```

shuffle

Shuffles (rearranges) elements for a given range by using a random number generator.

```
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void shuffle(RandomAccessIterator first,
    RandomAccessIterator last,
    UniformRandomNumberGenerator&& gen);
```

Parameters

first

An iterator to the first element in the range to be shuffled, inclusive. Must meet the requirements of

`RandomAccessIterator` and `ValueSwappable`.

last

An iterator to the last element in the range to be shuffled, exclusive. Must meet the requirements of

`RandomAccessIterator` and `ValueSwappable`.

gen

The random number generator that the `shuffle()` function will use for the operation. Must meet the requirements of a `UniformRandomNumberGenerator`.

Remarks

For more information, and a code sample that uses `shuffle()`, see [<random>](#).

sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Predicate>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Predicate comp);
```

Parameters

first

A random-access iterator addressing the position of the first element in the range to be sorted.

last

A random-access iterator addressing the position one past the final element in the range to be sorted.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. This binary predicate takes two arguments and returns **true** if the two arguments are in order and **false** otherwise. This comparator function must impose a strict weak ordering on pairs of elements from the sequence. For more information, see [Algorithms](#).

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm is not stable and so does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_sort` does preserve this original ordering.

The average of a sort complexity is $O(N \log N)$, where $N = last - first$.

Example

```

// alg_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
    {
        v1.push_back( 2 * ii + 1 );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order. specify binary predicate
    sort( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted (greater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    sort( v1.begin( ), v1.end( ), UDgreater );
    cout << "Resorted (UDgreater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

```

Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
Resorted (UDgreater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )

```

sort_heap

Converts a heap into a sorted range.

```
template<class RandomAccessIterator>
void sort_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Predicate>
void sort_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Predicate comp);
```

Parameters

first

A random-access iterator addressing the position of the first element in the target heap.

last

A random-access iterator addressing the position one past the final element in the target heap.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

After the application of this algorithm, the range it was applied to is no longer a heap.

This is not a stable sort because the relative order of equivalent elements is not necessarily preserved.

Heaps are an ideal way to implement priority queues and they are used in the implementation of the C++ Standard Library container adaptor [priority_queue Class](#).

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is at most $N \log N$, where $N = (last - first)$.

Example

```

// alg_sort_heap.cpp
// compile with: /EHsc
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>
#include <string>
#include <vector>
using namespace std;

void print(const string& s, const vector<int>& v) {
    cout << s << ": ( ";

    for (auto i = v.begin(); i != v.end(); ++i) {
        cout << *i << " ";
    }

    cout << ")" << endl;
}

int main() {
    vector<int> v;
    for (int i = 1; i <= 9; ++i) {
        v.push_back(i);
    }
    print("Initially", v);

    random_shuffle(v.begin(), v.end());
    print("After random_shuffle", v);

    make_heap(v.begin(), v.end());
    print("    After make_heap", v);

    sort_heap(v.begin(), v.end());
    print("    After sort_heap", v);

    random_shuffle(v.begin(), v.end());
    print("        After random_shuffle", v);

    make_heap(v.begin(), v.end(), greater<int>());
    print("After make_heap with greater<int>", v);

    sort_heap(v.begin(), v.end(), greater<int>());
    print("After sort_heap with greater<int>", v);
}

```

stable_partition

Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.

```

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Predicate pred );

```

Parameters

first

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

_Pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A predicate takes single argument and returns **true** or **false**.

Return Value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements a and b are equivalent, but not necessarily equal, if both $Pr(a, b)$ is false and $Pr(b, a)$ if false, where Pr is the parameter-specified predicate. The `stable_partition` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved. The algorithm `partition` does not necessarily preserve this original ordering.

Example

```
// alg_stable_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value ) {
    return value > 5;
}

int main() {
    using namespace std;
    vector <int> v1, v2;
    vector <int>::iterator Iter1, Iter2, result;

    int i;
    for ( i = 0 ; i <= 10 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 4 ; ii++ )
        v1.push_back( 5 );

    random_shuffle(v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Partition the range with predicate greater10
    result = stable_partition (v1.begin( ), v1.end( ), greater5 );
    cout << "The partitioned set of elements in v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "The first element in v1 to fail to satisfy the"
        << "\n predicate greater5 is: " << *result << "." << endl;
}
```


stable_sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.

```
template<class BidirectionalIterator>
void stable_sort( BidirectionalIterator first, BidirectionalIterator last );

template<class BidirectionalIterator, class BinaryPredicate>
void stable_sort(
    BidirectionalIterator first,
    BidirectionalIterator last,
    BinaryPredicate comp );
```

Parameters

first

A bidirectional iterator addressing the position of the first element in the range to be sorted.

last

A bidirectional iterator addressing the position one past the final element in the range to be sorted.

comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved.

The run-time complexity of `stable_sort` depends on the amount of memory available, but the best case (given sufficient memory) is $O(N \log N)$ and the worst case is $O(N (\log N)^2)$, where $N = last - first$. Usually, the `sort` algorithm is significantly faster than `stable_sort`.

Example

```

// alg_stable_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater (int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector <int> v1;
    vector <int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    stable_sort(v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order, specify binary predicate
    stable_sort(v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted (greater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    stable_sort(v1.begin( ), v1.end( ), UDgreater );
    cout << "Resorted (UDgreater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

```

Original vector v1 = ( 0 2 4 6 8 10 0 2 4 6 8 10 )
Sorted vector v1 = ( 0 0 2 2 4 4 6 6 8 8 10 10 )
Resorted (greater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
Resorted (UDgreater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )

```

swap

The first override exchanges the values of two objects. The second override exchanges the values between two

arrays of objects.

```
template<class Type>
    void swap(
        Type& left,
        Type& right);
template<class Type, size_t N>
    void swap(
        Type (& left)[N],
        Type (& right)[N]);\r
```

Parameters

left

For the first override, the first object to have its contents exchanged. For the second override, the first array of objects to have its contents exchanged.

right

For the first override, the second object to have its contents exchanged. For the second override, the second array of objects to have its contents exchanged.

Remarks

The first overload is designed to operate on individual objects. The second overload swaps the contents of objects between two arrays.

Example

```

// alg_swap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2, result;

    for ( int i = 0 ; i <= 10 ; i++ )
    {
        v1.push_back( i );
    }

    for ( int ii = 0 ; ii <= 4 ; ii++ )
    {
        v2.push_back( 5 );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    swap( v1, v2 );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;
}

```

```

Vector v1 is ( 0 1 2 3 4 5 6 7 8 9 10 ).
Vector v2 is ( 5 5 5 5 5 ).
Vector v1 is ( 5 5 5 5 5 ).
Vector v2 is ( 0 1 2 3 4 5 6 7 8 9 10 ).

```

swap_ranges

Exchanges the elements of one range with the elements of another, equal sized range.

```

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2 );

```

Parameters

first1

A forward iterator pointing to the first position of the first range whose elements are to be exchanged.

last1

A forward iterator pointing to one past the final position of the first range whose elements are to be exchanged.

first2

A forward iterator pointing to the first position of the second range whose elements are to be exchanged.

Return Value

A forward iterator pointing to one past the final position of the second range whose elements are to be exchanged.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation. The second range has to be as large as the first range.

The complexity is linear with *last1* - *first1* swaps performed. If elements from containers of the same type are being swapped, then the `swap` member function from that container should be used, because the member function typically has constant complexity.

Example

```

// alg_swap_ranges.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    deque<int> d1;
    vector<int>::iterator v1Iter1;
    deque<int>::iterator d1Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii =4 ; ii <= 9 ; ii++ )
    {
        d1.push_back( 6 );
    }

    cout << "Vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "Deque d1 is ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;

    swap_ranges ( v1.begin( ), v1.end( ), d1.begin( ) );

    cout << "After the swap_range, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "After the swap_range deque d1 is ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;
}

```

```

Vector v1 is ( 0 1 2 3 4 5 ).
Deque d1 is ( 6 6 6 6 6 6 ).
After the swap_range, vector v1 is ( 6 6 6 6 6 6 ).
After the swap_range deque d1 is ( 0 1 2 3 4 5 ).

```

transform

Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.

```

template<class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(
    InputIterator first1,
    InputIterator last1,
    OutputIterator result,
    UnaryFunction func );

template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryFunction>
OutputIterator transform(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    OutputIterator result,
    BinaryFunction func );

```

Parameters

first1

An input iterator addressing the position of the first element in the first source range to be operated on.

last1

An input iterator addressing the position one past the final element in the first source range operated on.

first2

An input iterator addressing the position of the first element in the second source range to be operated on.

result

An output iterator addressing the position of the first element in the destination range.

_Func

User-defined unary function object used in the first version of the algorithm that is applied to each element in the first source range or A user-defined (UD) binary function object used in the second version of the algorithm that is applied pairwise, in a forward order, to the two source ranges.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation. The destination range must be large enough to contain the transformed source range.

If *result* is set equal to *first1* in the first version of the algorithm, then the source and destination ranges will be the same and the sequence will be modified in place. But the *result* may not address a position within the range [`first1 + 1, last1`).

The complexity is linear with at most (`last1 - first1`) comparisons.

Example

```

// alg_transform.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue

```

```

{
    private:
        Type Factor;    // The value to multiply by
    public:
        // Constructor initializes the value to multiply by
        MultValue ( const Type& val ) : Factor ( val ) {
        }

        // The function call for the element to be multiplied
        Type operator( ) ( Type& elem ) const
        {
            return elem * Factor;
        }
};

int main()
{
    using namespace std;
    vector<int> v1, v2 ( 7 ), v3 ( 7 );
    vector<int>::iterator Iter1, Iter2 , Iter3;

    // Constructing vector v1
    int i;
    for ( i = -4 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Original vector  v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Modifying the vector v1 in place
    transform (v1.begin( ), v1.end( ), v1.begin( ), MultValue<int> ( 2 ) );
    cout << "The elements of the vector v1 multiplied by 2 in place gives:"
        << "\n v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Using transform to multiply each element by a factor of 5
    transform ( v1.begin( ), v1.end( ), v2.begin( ), MultValue<int> ( 5 ) );

    cout << "Multiplying the elements of the vector v1mod\n "
        << "by the factor 5 & copying to v2 gives:\n v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // The second version of transform used to multiply the
    // elements of the vectors v1mod & v2 pairwise
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        multiplies<int>( ) );

    cout << "Multiplying elements of the vectors v1mod and v2 pairwise "
        << "gives:\n v3 = ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}

```



```
Original vector v1 = ( -4 -3 -2 -1 0 1 2 ).
The elements of the vector v1 multiplied by 2 in place gives:
v1mod = ( -8 -6 -4 -2 0 2 4 ).
Multiplying the elements of the vector v1mod
by the factor 5 & copying to v2 gives:
v2 = ( -40 -30 -20 -10 0 10 20 ).
Multiplying elements of the vectors v1mod and v2 pairwise gives:
v3 = ( 320 180 80 20 0 20 80 ).
```

unique

Removes duplicate elements that are adjacent to each other in a specified range.

```
template<class ForwardIterator>
ForwardIterator unique(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class Predicate>
ForwardIterator unique(
    ForwardIterator first,
    ForwardIterator last,
    Predicate comp);
```

Parameters

first

A forward iterator addressing the position of the first element in the range to be scanned for duplicate removal.

last

A forward iterator addressing the position one past the final element in the range to be scanned for duplicate removal.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator to the new end of the modified sequence that contains no consecutive duplicates, addressing the position one past the last element not removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation. The number of elements in the sequence is not changed by the algorithm `unique` and the elements beyond the end of the modified sequence are dereferenceable but not specified.

The complexity is linear, requiring $(\text{last} - \text{first}) - 1$ comparisons.

List provides a more efficient member function "unique", which may perform better.

These algorithms cannot be used on an associative container.

Example

```

// alg_unique.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>

using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 == elem2;
};

int main()
{
    vector <int> v1;
    vector <int>::iterator v1_Iter1, v1_Iter2, v1_Iter3,
        v1_NewEnd1, v1_NewEnd2, v1_NewEnd3;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( 5 );
        v1.push_back( -5 );
    }

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
    {
        v1.push_back( 4 );
    }
    v1.push_back( 7 );

    cout << "Vector v1 is ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    // Remove consecutive duplicates
    v1_NewEnd1 = unique ( v1.begin( ), v1.end( ) );

    cout << "Removing adjacent duplicates from vector v1 gives\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    // Remove consecutive duplicates under the binary predicate mod_equals
    v1_NewEnd2 = unique ( v1.begin( ), v1_NewEnd1 , mod_equal );

    cout << "Removing adjacent duplicates from vector v1 under the\n "
        << " binary predicate mod_equal gives\n ( " ;
    for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
        cout << *v1_Iter2 << " ";
    cout << ")." << endl;

    // Remove elements if preceded by an element that was greater
    v1_NewEnd3 = unique ( v1.begin( ), v1_NewEnd2, greater<int>( ) );

    cout << "Removing adjacent elements satisfying the binary\n "
        << " predicate mod_equal from vector v1 gives ( " ;
    for ( v1_Iter3 = v1.begin( ) ; v1_Iter3 != v1_NewEnd3 ; v1_Iter3++ )
        cout << *v1_Iter3 << " ";

```

```
    cout << ")." << endl;
}
```

Vector v1 is (5 -5 5 -5 5 -5 5 -5 4 4 4 7).
Removing adjacent duplicates from vector v1 gives
(5 -5 5 -5 5 -5 5 -5 4 7).
Removing adjacent duplicates from vector v1 under the
binary predicate mod_equal gives
(5 4 7).
Removing adjacent elements satisfying the binary
predicate mod_equal from vector v1 gives (5 7).

unique_copy

Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy( InputIterator first,
    InputIterator last,
    OutputIterator result );

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy( InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryPredicate comp );
```

Parameters

first

A forward iterator addressing the position of the first element in the source range to be copied.

last

A forward iterator addressing the position one past the final element in the source range to be copied.

result

An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.

comp

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.

The ranges referenced must be valid; all pointers must be dereferenceable and within a sequence the last position is reachable from the first by incrementation.

The complexity is linear, requiring (`last` - `first`) comparisons.

Example

```

// alg_unique_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>

using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 ) {
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 == elem2;
};

int main() {
    vector<int> v1;
    vector<int>::iterator v1_Iter1, v1_Iter2,
        v1_NewEnd1, v1_NewEnd2;

    int i;
    for ( i = 0 ; i <= 1 ; i++ ) {
        v1.push_back( 5 );
        v1.push_back( -5 );
    }

    int ii;
    for ( ii = 0 ; ii <= 2 ; ii++ )
        v1.push_back( 4 );
    v1.push_back( 7 );

    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
        v1.push_back( 10 );

    cout << "Vector v1 is\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    // Copy first half to second, removing consecutive duplicates
    v1_NewEnd1 = unique_copy ( v1.begin( ), v1.begin( ) + 8, v1.begin( ) + 8 );

    cout << "Copying the first half of the vector to the second half\n "
        << "while removing adjacent duplicates gives\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    int iv;
    for ( iv = 0 ; iv <= 7 ; iv++ )
        v1.push_back( 10 );

    // Remove consecutive duplicates under the binary predicate mod_equals
    v1_NewEnd2 = unique_copy ( v1.begin( ), v1.begin( ) + 14,
        v1.begin( ) + 14 , mod_equal );

    cout << "Copying the first half of the vector to the second half\n "
        << " removing adjacent duplicates under mod_equals gives\n ( " ;
    for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
        cout << *v1_Iter2 << " ";
    cout << ")." << endl;
}

```

upper_bound

Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator, class Type>
ForwardIterator upper_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ForwardIterator, class Type, class Predicate>
ForwardIterator upper_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    Predicate comp);
```

Parameters

first

The position of the first element in the range to be searched.

last

The position one past the final element in the range to be searched.

value

The value in the ordered range that needs to be exceeded by the value of the element addressed by the iterator returned.

comp

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return Value

A forward iterator to the position of the first element that has a value greater than a specified value.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of the use of `upper_bound` and where the ordering criterion is the same as specified by the binary predicate.

The range is not modified by `upper_bound`.

The value types of the forward iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to `(last - first)`.

Example

```
// alg_upper_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
```

```

#include <functional>      // greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    vector<int> v1;
    // Constructing vector v1 with default less-than ordering
    for ( auto i = -1 ; i <= 4 ; ++i )
    {
        v1.push_back( i );
    }

    for ( auto ii = -3 ; ii <= 0 ; ++ii )
    {
        v1.push_back( ii );
    }

    cout << "Starting vector v1 = ( " ;
    for (const auto &Iter : v1)
        cout << Iter << " ";
    cout << ")." << endl;

    sort(v1.begin(), v1.end());
    cout << "Original vector v1 with range sorted by the\n "
        << "binary predicate less than is v1 = ( " ;
    for (const auto &Iter : v1)
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vector v2 with range sorted by greater
    vector<int> v2(v1);

    sort(v2.begin(), v2.end(), greater<int>());

    cout << "Original vector v2 with range sorted by the\n "
        << "binary predicate greater is v2 = ( " ;
    for (const auto &Iter : v2)
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vectors v3 with range sorted by mod_lesser
    vector<int> v3(v1);
    sort(v3.begin(), v3.end(), mod_lesser);

    cout << "Original vector v3 with range sorted by the\n "
        << "binary predicate mod_lesser is v3 = ( " ;
    for (const auto &Iter : v3)
        cout << Iter << " ";
    cout << ")." << endl;

    // Demonstrate upper_bound

    vector<int>::iterator Result;

    // upper_bound of 3 in v1 with default binary predicate less<int>()
    Result = upper_bound(v1.begin(), v1.end(), 3);
    cout << "The upper_bound in v1 for the element with a value of 3 is: "

```

```

        << *Result << "." << endl;

// upper_bound of 3 in v2 with the binary predicate greater<int>( )
Result = upper_bound(v2.begin(), v2.end(), 3, greater<int>());
cout << "The upper_bound in v2 for the element with a value of 3 is: "
    << *Result << "." << endl;

// upper_bound of 3 in v3 with the binary predicate mod_lesser
Result = upper_bound(v3.begin(), v3.end(), 3, mod_lesser);
cout << "The upper_bound in v3 for the element with a value of 3 is: "
    << *Result << "." << endl;
}

```

See also

[<algorithm>](#)

<allocators>

10/31/2018 • 5 minutes to read • [Edit Online](#)

Defines several templates that help allocate and free memory blocks for node-based containers.

Syntax

```
#include <allocators>
```

Remarks

The <allocators> header provides six allocator templates that can be used to select memory-management strategies for node-based containers. For use with these templates, it also provides several different synchronization filters to tailor the memory-management strategy to a variety of different multithreading schemes (including none). Matching a memory management strategy to the known memory usage patterns, and synchronization requirements, of a particular application can often increase the speed or reduce the overall memory requirements of an application.

The allocator templates are implemented with reusable components that can be customized or replaced to provide additional memory-management strategies.

The node-based containers in the C++ Standard Library (std::list, std::set, std::multiset, std::map and std::multimap) store their elements in individual nodes. All the nodes for a particular container type are the same size, so the flexibility of a general-purpose memory manager is not needed. Because the size of each memory block is known at compile time, the memory manager can be much simpler and faster.

When used with containers that are not node-based (such as the C++ Standard Library containers std::vector, std::deque, and std::basic_string), the allocator templates will work correctly, but are not likely to provide any performance improvement over the default allocator.

An allocator is a template class that describes an object that manages storage allocation and freeing for objects and arrays of objects of a designated type. Allocator objects are used by several container template classes in the C++ Standard Library.

The allocators are all templates of this type:

```
template<class Type>  
class allocator;
```

where the template argument `Type` is the type managed by the allocator instance. The C++ Standard Library provides a default allocator, template class `allocator`, which is defined in [<memory>](#). The <allocators> header provides the following allocators:

- [allocator_newdel](#)
- [allocator_unbounded](#)
- [allocator_fixed_size](#)
- [allocator_variable_size](#)
- [allocator_suballoc](#)

- [allocator_chunklist](#)

Use an appropriate instantiation of an allocator as the second type argument when creating a container, such as the following code example.

```
#include <list>
#include <allocators>
std::list<int, stdext::allocators::allocator_chunklist<int> > _List0;
```

_List0 allocates nodes with `allocator_chunklist` and the default synchronization filter.

Use the macro `ALLOCATOR_DECL` to create allocator templates with synchronization filters other than the default:

```
#include <list>
#include <allocators>
ALLOCATOR_DECL(CACHE_CHUNKLIST, stdext::allocators::sync_per_thread, Alloc);
std::list<int, alloc<int> > _List1;
```

_List1 allocates nodes with `allocator_chunklist` and the `sync_per_thread` synchronization filter.

A block allocator is a cache or a filter. A cache is a template class that takes one argument of type `std::size_t`. It defines a block allocator that allocates and deallocates memory blocks of a single size. It must obtain memory using operator **new**, but it need not make a separate call to operator **new** for each block. It may, for example, suballocate from a larger block or cache deallocated blocks for subsequent reallocation.

With a compiler that cannot compile rebind the value of the `std::size_t` argument used when the template was instantiated is not necessarily the value of the argument `_Sz` passed to a cache's member functions `allocate` and `deallocate`.

`<allocators>` provides the following cache templates:

- [cache_freelist](#)
- [cache_suballoc](#)
- [cache_chunklist](#)

A filter is a block allocator that implements its member functions using another block allocator which is passed to it as a template argument. The most common form of filter is a synchronization filter, which applies a synchronization policy to control access to the member functions of an instance of another block allocator.

`<allocators>` provides the following synchronization filters:

- [sync_none](#)
- [sync_per_container](#)
- [sync_per_thread](#)
- [sync_shared](#)

`<allocators>` also provides the filter `rts_alloc`, which holds multiple block allocator instances and determines which instance to use for allocation or deallocation at runtime instead of at compile time. It is used with compilers that cannot compile rebind.

A synchronization policy determines how an allocator instance handles simultaneous allocation and deallocation requests from multiple threads. The simplest policy is to pass all requests directly through to the underlying cache object, leaving synchronization management to the user. A more complex policy could be to use a mutex to serialize access to the underlying cache object.

If a compiler supports compiling both single-threaded and multi-threaded applications, the default synchronization filter for single-threaded applications is `sync_none`; for all other cases it is `sync_shared`.

The cache template `cache_freelist` takes a max class argument which determines the maximum number of elements to be stored in the free list.

<allocators> provides the following max classes:

- [max_none](#)
- [max_unbounded](#)
- [max_fixed_size](#)
- [max_variable_size](#)

Macros

MACRO	DESCRIPTION
ALLOCATOR_DECL	Yields an allocator template class.
CACHE_CHUNKLIST	Yields <code>stdext::allocators::cache_chunklist<sizeof(Type)></code> .
CACHE_FREELIST	Yields <code>stdext::allocators::cache_freelist<sizeof(Type), max></code> .
CACHE_SUBALLOC	Yields <code>stdext::allocators::cache_suballoc<sizeof(Type)></code> .
SYNC_DEFAULT	Yields a synchronization filter.

Operators

OPERATOR	DESCRIPTION
operator!= (<allocators>)	Tests for inequality between allocator objects of a specified class.
operator== (<allocators>)	Tests for equality between allocator objects of a specified class.

Classes

CLASS	DESCRIPTION
allocator_base	Defines the base class and common functions needed to create a user-defined allocator from a synchronization filter.
allocator_chunklist	Describes an object that manages storage allocation and freeing for objects using a cache of type cache_chunklist .
allocator_fixed_size	Describes an object that manages storage allocation and freeing for objects of type <code>Type</code> using a cache of type cache_freelist with a length managed by max_fixed_size .

CLASS	DESCRIPTION
allocator_newdel	Implements an allocator that uses operator delete to deallocate a memory block and operator new to allocate a memory block.
allocator_suballoc	Describes an object that manages storage allocation and freeing for objects of type <code>Type</code> using a cache of type cache_suballoc .
allocator_unbounded	Describes an object that manages storage allocation and freeing for objects of type <code>Type</code> using a cache of type cache_freelist with a length managed by max_unbounded .
allocator_variable_size	Describes an object that manages storage allocation and freeing for objects of type <code>Type</code> using a cache of type cache_freelist with a length managed by max_variable_size .
cache_chunklist	Defines a block allocator that allocates and deallocates memory blocks of a single size.
cache_freelist	Defines a block allocator that allocates and deallocates memory blocks of a single size.
cache_suballoc	Defines a block allocator that allocates and deallocates memory blocks of a single size.
freelist	Manages a list of memory blocks.
max_fixed_size	Describes a max class object that limits a freelist object to a fixed maximum length.
max_none	Describes a max class object that limits a freelist object to a maximum length of zero.
max_unbounded	Describes a max class object that does not limit the maximum length of a freelist object.
max_variable_size	Describes a max class object that limits a freelist object to a maximum length that is roughly proportional to the number of allocated memory blocks.
rts_alloc	The <code>rts_alloc</code> template class describes a filter that holds an array of cache instances and determines which instance to use for allocation and deallocation at runtime instead of at compile time.
sync_none	Describes a synchronization filter that provides no synchronization.
sync_per_container	Describes a synchronization filter that provides a separate cache object for each allocator object.
sync_per_thread	Describes a synchronization filter that provides a separate cache object for each thread.

CLASS	DESCRIPTION
sync_shared	Describes a synchronization filter that uses a mutex to control access to a cache object that is shared by all allocators.

Requirements

Header: <allocators>

Namespace: stdext

See also

[Header Files Reference](#)

<allocators> macros

10/31/2018 • 2 minutes to read • [Edit Online](#)

ALLOCATOR_DECL	CACHE_CHUNKLIST	CACHE_FREELIST
CACHE_SUBALLOC	SYNC_DEFAULT	

ALLOCATOR_DECL

Yields an allocator template class.

```
#define ALLOCATOR_DECL(cache, sync, name) <alloc_template>
```

Remarks

The macro yields a template definition `template <class Type> class name {.....}` and a specialization `template <> class name<void> {.....}` which together define an allocator template class that uses the synchronization filter `sync` and a cache of type `cache`.

For compilers that can compile `rebind`, the resulting template definition looks like this:

```
struct rebind
{
    /* convert a name<Type> to a name<Other> */
    typedef name<Other> other;
};
```

For compilers that cannot compile `rebind` the resulting template definition looks like this:

```
template <class Type> class name
: public stdext::allocators::allocator_base<Type,
    sync<stdext::allocators::rts_alloc<cache>>>
{
public:
    name() {}
    template <class Other>
    name(const name<Other>&) {}
    template <class Other>
    name& operator= (const name<Other>&)
    {
        return *this;
    }
};
```

CACHE_CHUNKLIST

Yields `stdext::allocators::cache_chunklist<sizeof(Type)>`.

```
#define CACHE_CHUNKLIST <cache_class>
```

Remarks

CACHE_FREELIST

Yields `stdext::allocators::cache_freelist<sizeof(Type), max>` .

```
#define CACHE_FREELIST(max) <cache_class>
```

Remarks

CACHE_SUBALLOC

Yields `stdext::allocators::cache_suballoc<sizeof(Type)>` .

```
#define CACHE_SUBALLOC <cache_class>
```

Remarks

SYNC_DEFAULT

Yields a synchronization filter.

```
#define SYNC_DEFAULT <sync_template>
```

Remarks

If a compiler supports compiling both single-threaded and multi-threaded applications, for single-threaded applications the macro yields `stdext::allocators::sync_none` ; in all other cases it yields

`stdext::allocators::sync_shared` .

See also

[<allocators>](#)

<allocators> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

These are the global template operator functions defined in <allocators>. For class member operator functions, see the class documentation.

operator!=	operator==

operator!=

Tests for inequality between allocator objects of a specified class.

```
template <class Type, class Sync>
bool operator!=(
    const allocator_base<Type, Sync>& left,
    const allocator_base<Type, Sync>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	One of the allocator objects to be tested for inequality.
<i>right</i>	One of the allocator objects to be tested for inequality.

Return Value

true if the allocator objects are not equal; **false** if allocator objects are equal.

Remarks

The template operator returns `!(left == right)`.

operator==

Tests for equality between allocator objects of a specified class.

```
template <class Type, class Sync>
bool operator==(
    const allocator_base<Type, Sync>& left,
    const allocator_base<Type, Sync>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	One of the allocator objects to be tested for equality.
<i>right</i>	One of the allocator objects to be tested for equality.

Return Value

true if the allocator objects are equal; **false** if allocator objects are not equal.

Remarks

This template operator returns `left.equals(right)`.

See also

[<allocators>](#)

allocator_base Class

10/31/2018 • 5 minutes to read • [Edit Online](#)

Defines the base class and common functions needed to create a user-defined allocator from a synchronization filter.

Syntax

```
template <class Type, class Sync>
class allocator_base
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.
<i>Sync</i>	The synchronization policy for the allocator, which is sync_none Class , sync_per_container Class , sync_per_thread Class , or sync_shared Class .

Constructors

CONSTRUCTOR	DESCRIPTION
allocator_base	Constructs an object of type <code>allocator_base</code> .

Typedefs

TYPE NAME	DESCRIPTION
const_pointer	A type that provides a constant pointer to the type of object managed by the allocator.
const_reference	A type that provides a constant reference to type of object managed by the allocator.
difference_type	A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.
pointer	A type that provides a pointer to the type of object managed by the allocator.
reference	A type that provides a reference to the type of object managed by the allocator.
size_type	An unsigned integral type that can represent the length of any sequence that an object of template class <code>allocator_base</code> can allocate.

TYPE NAME	DESCRIPTION
<code>value_type</code>	A type that is managed by the allocator.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>_Charalloc</code>	Allocates storage for an array of type char .
<code>_Chardealloc</code>	Frees storage for the array containing elements of type char .
<code>address</code>	Finds the address of an object whose value is specified.
<code>allocate</code>	Allocates a block of memory large enough to store at least some specified number of elements.
<code>construct</code>	Constructs a specific type of object at a specified address that is initialized with a specified value.
<code>deallocate</code>	Frees a specified number of objects from storage beginning at a specified position.
<code>destroy</code>	Calls an objects destructor without deallocating the memory where the object was stored.
<code>max_size</code>	Returns the number of elements of type <i>Type</i> that could be allocated by an object of class allocator before the free memory is used up.

Requirements

Header: <allocators>

Namespace: stdext

allocator_base::_Charalloc

Allocates storage for an array of type **char**.

```
char *_Charalloc(size_type count);
```

Parameters

PARAMETER	DESCRIPTION
<code>count</code>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

This member function is used by containers when compiled with a compiler that cannot compile rebind. It implements `_Charalloc` for the user-defined allocator by returning the result of a call to the `allocate` function of

the synchronization filter.

allocator_base::_Chardealloc

Frees storage for the array containing elements of type **char**.

```
void _Chardealloc(void* ptr, size_type count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

This member function is used by containers when compiled with a compiler that cannot compile rebind. It implements `_Chardealloc` for the user-defined allocator by calling the `deallocate` function of the synchronization filter. The pointer `ptr` must have been earlier returned by a call to `_Charalloc` for an allocator object that compares equal to `*this`, allocating an array object of the same size and type. `_Chardealloc` never throws an exception.

allocator_base::address

Finds the address of an object whose value is specified.

```
pointer address(reference val);

const_pointer address(const_reference val);
```

Parameters

val

The const or nonconst value of the object whose address is being searched for.

Return Value

A const or nonconst pointer to the object found of, respectively, const or nonconst value.

Remarks

This member function is implemented for the user-defined allocator by returning `&val`.

allocator_base::allocate

Allocates a block of memory large enough to store at least some specified number of elements.

```
template <class Other>
pointer allocate(size_type _Nx, const Other* _Hint = 0);

pointer allocate(size_type _Nx);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The number of elements in the array to be allocated.
<code>_Hint</code>	This parameter is ignored.

Return Value

A pointer to the allocated object.

Remarks

The member function implements memory allocation for the user-defined allocator by returning the result of a call to the `allocate` function of the synchronization filter of type `Type *` if `_Nx == 1`, otherwise by returning the result of a call to `operator new(_Nx * sizeof(Type))` cast to `Type *`.

allocator_base::allocator_base

Constructs an object of type `allocator_base`.

```
allocator_base();

template <class Other>
allocator_base(const allocator_base<Other, Sync>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The allocator object to be copied.

Remarks

The first constructor constructs an `allocator_base` instance. The second constructor constructs an `allocator_base` instance such that for any `allocator_base<Type, _Sync>` instance `a`,
`allocator_base<Type, Sync>(allocator_base<Other, Sync>(a)) == a`.

allocator_base::const_pointer

A type that provides a constant pointer to the type of object managed by the allocator.

```
typedef const Type *const_pointer;
```

allocator_base::const_reference

A type that provides a constant reference to type of object managed by the allocator.

```
typedef const Type& const_reference;
```

allocator_base::construct

Constructs a specific type of object at a specified address that is initialized with a specified value.

```
void construct(pointer ptr, const Type& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the location where the object is to be constructed.
<i>val</i>	The value with which the object being constructed is to be initialized.

Remarks

This member function is implemented for the user-defined allocator by calling `new((void*)ptr Type(val))`.

allocator_base::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(pointer ptr, size_type _Nx);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>_Nx</i>	The number of objects to be deallocated from storage.

Remarks

This member function is implemented for the user-defined allocator by calling `deallocate(ptr)` on the synchronization filter `Sync` if `_Nx == 1`, otherwise by calling `operator delete(_Nx * ptr)`.

allocator_base::destroy

Calls an objects destructor without deallocating the memory where the object was stored.

```
void destroy(pointer ptr);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer designating the address of the object to be destroyed.

Remarks

This member function is implemented for the user-defined allocator by calling `ptr->~Type()`.

allocator_base::difference_type

A signed integral type that can represent the difference between values of pointers to the type of object managed

by the allocator.

```
typedef std::ptrdiff_t difference_type;
```

allocator_base::max_size

Returns the number of elements of type `Type` that could be allocated by an object of class allocator before the free memory is used up.

```
size_type max_size() const;
```

Return Value

The number of elements that could be allocated.

Remarks

This member function is implemented for the user-defined allocator by returning `(size_t)-1 / sizeof(Type)` if `0 < (size_t)-1 / sizeof(Type)`, otherwise `1`.

allocator_base::pointer

A type that provides a pointer to the type of object managed by the allocator.

```
typedef Type *pointer;
```

allocator_base::reference

A type that provides a reference to the type of object managed by the allocator.

```
typedef Type& reference;
```

allocator_base::size_type

An unsigned integral type that can represent the length of any sequence that an object of template class `allocator_base` can allocate.

```
typedef std::size_t size_type;
```

allocator_base::value_type

A type that is managed by the allocator.

```
typedef Type value_type;
```

See also

[<allocators>](#)

allocator_chunklist Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that manages storage allocation and freeing for objects using a cache of type [cache_chunklist](#).

Syntax

```
template <class Type>
class allocator_chunklist;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_CHUNKLIST, SYNC_DEFAULT, allocator_chunklist);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

allocator_fixed_size Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type [cache_freelist](#) with a length managed by [max_fixed_size](#).

Syntax

```
template <class Type>
class allocator_fixed_size;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_fixed_size<10>), SYNC_DEFAULT, allocator_fixed_size);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

allocator_newdel Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Implements an allocator that uses **operator delete** to deallocate a memory block and **operator new** to allocate a memory block.

Syntax

```
template <class Type>
class allocator_newdel;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_FREELIST stdext::allocators::max_none), SYNC_DEFAULT, allocator_newdel);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

allocator_suballoc Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type [cache_suballoc](#).

Syntax

```
template <class Type>
class allocator_suballoc;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_SUBALLOC, SYNC_DEFAULT, allocator_suballoc);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

allocator_unbounded Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type [cache_freelist](#) with a length managed by [max_unbounded](#).

Syntax

```
template <class Type>
class allocator_unbounded;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_unbounded), SYNC_DEFAULT, allocator_unbounded);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

allocator_variable_size Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type [cache_freelist](#) with a length managed by [max_variable_size](#).

Syntax

```
template <class Type>
class allocator_variable_size;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of elements allocated by the allocator.

Remarks

The [ALLOCATOR_DECL](#) macro passes this class as the *name* parameter in the following statement:

```
ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_variable_size), SYNC_DEFAULT, allocator_variable_size);
```

Requirements

Header: <allocators>

Namespace: stdext

See also

[<allocators>](#)

cache_chunklist Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a [block allocator](#) that allocates and deallocates memory blocks of a single size.

Syntax

```
template <std::size_t Sz, std::size_t Nelts = 20>
class cache_chunklist
```

Parameters

PARAMETER	DESCRIPTION
<code>Sz</code>	The number of elements in the array to be allocated.

Remarks

This template class uses **operator new** to allocate chunks of raw memory, suballocating blocks to allocate storage for a memory block when needed; it stores deallocated memory blocks in a separate free list for each chunk, and uses **operator delete** to deallocate a chunk when none of its memory blocks is in use.

Each memory block holds `Sz` bytes of usable memory and a pointer to the chunk that it belongs to. Each chunk holds `Nelts` memory blocks, three pointers, an int and the data that **operator new** and **operator delete** require.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>cache_chunklist</code>	Constructs an object of type <code>cache_chunklist</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>allocate</code>	Allocates a block of memory.
<code>deallocate</code>	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_chunklist::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_chunklist::cache_chunklist

Constructs an object of type `cache_chunklist`.

```
cache_chunklist();
```

Remarks

cache_chunklist::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

See also

[<allocators>](#)

cache_freelist Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a [block allocator](#) that allocates and deallocates memory blocks of a single size.

Syntax

```
template <std::size_t Sz, class Max>
class cache_freelist
```

Parameters

PARAMETER	DESCRIPTION
<i>Sz</i>	The number of elements in the array to be allocated.
<i>Max</i>	The max class representing the maximum size of the free list. This can be max_fixed_size , max_none , max_unbounded , or max_variable_size .

Remarks

The `cache_freelist` template class maintains a free list of memory blocks of size *Sz*. When the free list is full it uses **operator delete** to deallocate memory blocks. When the free list is empty it uses **operator new** to allocate new memory blocks. The maximum size of the free list is determined by the class max class passed in the *Max* parameter.

Each memory block holds *Sz* bytes of usable memory and the data that **operator new** and **operator delete** require.

Constructors

CONSTRUCTOR	DESCRIPTION
cache_freelist	Constructs an object of type <code>cache_freelist</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_freelist::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_freelist::cache_freelist

Constructs an object of type `cache_freelist`.

```
cache_freelist();
```

Remarks

cache_freelist::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

See also

[<allocators>](#)

cache_suballoc Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a [block allocator](#) that allocates and deallocates memory blocks of a single size.

Syntax

```
template <std::size_t Sz, size_t Nelts = 20>
class cache_suballoc
```

Parameters

PARAMETER	DESCRIPTION
Sz	The number of elements in the array to be allocated.

Remarks

The cache_suballoc template class stores deallocated memory blocks in a free list with unbounded length, using `freelist<sizeof(Type), max_unbounded>`, and suballocates memory blocks from a larger chunk allocated with **operator new** when the free list is empty.

Each chunk holds `Sz * Nelts` bytes of usable memory and the data that **operator new** and **operator delete** require. Allocated chunks are never freed.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>cache_suballoc</code>	Constructs an object of type <code>cache_suballoc</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>allocate</code>	Allocates a block of memory.
<code>deallocate</code>	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_suballoc::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_suballoc::cache_suballoc

Constructs an object of type `cache_suballoc`.

```
cache_suballoc();
```

Remarks

cache_suballoc::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

See also

[<allocators>](#)

freelist Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Manages a list of memory blocks.

Syntax

```
template <std::size_t Sz, class Max>
class freelist : public Max
```

Parameters

PARAMETER	DESCRIPTION
<i>Sz</i>	The number of elements in the array to be allocated.
<i>Max</i>	The max class representing the maximum number of elements to be stored in the free list. The max class can be max_none , max_unbounded , max_fixed_size , or max_variable_size .

Remarks

This template class manages a list of memory blocks of size *Sz* with the maximum length of the list determined by the max class passed in *Max*.

Constructors

CONSTRUCTOR	DESCRIPTION
freelist	Constructs an object of type <code>freelist</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
pop	Removes the first memory block from the free list.
push	Adds a memory block to the list.

Requirements

Header: <allocators>

Namespace: stdext

freelist::freelist

Constructs an object of type `freelist`.

```
freelist();
```

Remarks

freelist::pop

Removes the first memory block from the free list.

```
void *pop();
```

Return Value

Returns a pointer to the memory block removed from the list.

Remarks

The member function returns NULL if the list is empty. Otherwise, it removes the first memory block from the list.

freelist::push

Adds a memory block to the list.

```
bool push(void* ptr);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the memory block to be added to the free list.

Return Value

true if the `full` function of the max class returns **false**; otherwise, the `push` function returns **false**.

Remarks

If the `full` function of the max class returns **false**, this member function adds the memory block pointed to by *ptr* to the head of the list.

See also

[<allocators>](#)

max_fixed_size Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [max class](#) object that limits a [freelist](#) object to a fixed maximum length.

Syntax

```
template <std::size_t Max>
class max_fixed_size
```

Parameters

PARAMETER	DESCRIPTION
<i>Max</i>	The max class that determines the maximum number of elements to store in the <code>freelist</code> .

Constructors

CONSTRUCTOR	DESCRIPTION
<code>max_fixed_size</code>	Constructs an object of type <code>max_fixed_size</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>allocated</code>	Increments the count of allocated memory blocks.
<code>deallocated</code>	Decrements the count of allocated memory blocks.
<code>full</code>	Returns a value that specifies whether more memory blocks should be added to the free list.
<code>released</code>	Decrements the count of memory blocks on the free list.
<code>saved</code>	Increments the count of memory blocks on the free list.

Requirements

Header: `<allocators>`

Namespace: `stdext`

max_fixed_size::allocated

Increments the count of allocated memory blocks.

```
void allocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

The member function does nothing. This member function is called after each successful call by `cache_freelist::allocate` to operator **new**. The argument `_Nx` is the number of memory blocks in the chunk allocated by operator **new**.

`max_fixed_size::deallocated`

Decrements the count of allocated memory blocks.

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator **delete**. The argument `_Nx` is the number of memory blocks in the chunk deallocated by operator **delete**.

`max_fixed_size::full`

Returns a value that specifies whether more memory blocks should be added to the free list.

```
bool full();
```

Return Value

true if `Max <= _Nblocks`; otherwise, **false**.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns **true**, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator **delete** to deallocate the block.

`max_fixed_size::max_fixed_size`

Constructs an object of type `max_fixed_size`.

```
max_fixed_size();
```

Remarks

This constructor initializes the stored value `_Nblocks` to zero.

`max_fixed_size::released`

Decrements the count of memory blocks on the free list.

```
void released();
```

Remarks

Decrements the stored value `_Nblocks`. The `released` member function of the current [max class](#) is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_fixed_size::saved

Increments the count of memory blocks on the free list.

```
void saved();
```

Remarks

This member function increments the stored value `_Nblocks`. This member function is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

[<allocators>](#)

max_none Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [max class](#) object that limits a [freelist](#) object to a maximum length of zero.

Syntax

```
template <std::size_t Max>
class max_none
```

Parameters

PARAMETER	DESCRIPTION
<i>Max</i>	The max class that determines the maximum number of elements to store in the <code>freelist</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_none::allocated

Increments the count of allocated memory blocks.

```
void allocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<i>_Nx</i>	The increment value.

Remarks

This member function does nothing. It is called after each successful call by `cache_freelist::allocate` to operator **new**. The argument `_Nx` is the number of memory blocks in the chunk allocated by operator **new**.

max_none::deallocated

Decrements the count of allocated memory blocks.

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator **delete**. The argument `_Nx` is the number of memory blocks in the chunk deallocated by operator **delete**.

max_none::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
bool full();
```

Return Value

This member function always returns **true**.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns **true**, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator **delete** to deallocate the block.

max_none::released

Decrements the count of memory blocks on the free list.

```
void released();
```

Remarks

This member function does nothing. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_none::saved

Increments the count of memory blocks on the free list.

```
void saved();
```

Remarks

This member function does nothing. It is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

[<allocators>](#)

max_unbounded Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [max class](#) object that does not limit the maximum length of a [freelist](#) object.

Syntax

```
class max_unbounded
```

Member functions

MEMBER FUNCTION	DESCRIPTION
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_unbounded::allocated

Increments the count of allocated memory blocks.

```
void allocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

This member function does nothing. It is called after each successful call by `cache_freelist::allocate` to operator **new**. The argument `_Nx` is the number of memory blocks in the chunk allocated by operator **new**.

max_unbounded::deallocated

Decrements the count of allocated memory blocks.

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator **delete**. The argument `_Nx` is the number of memory blocks in the chunk deallocated by operator **delete**.

max_unbounded::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
bool full();
```

Return Value

The member function always returns **false**.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns **true**, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator **delete** to deallocate the block.

max_unbounded::released

Decrements the count of memory blocks on the free list.

```
void released();
```

Remarks

This member function does nothing. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_unbounded::saved

Increments the count of memory blocks on the free list.

```
void saved();
```

Remarks

This member function does nothing. It is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

[<allocators>](#)

max_variable_size Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [max class](#) object that limits a [freelist](#) object to a maximum length that is roughly proportional to the number of allocated memory blocks.

Syntax

```
class max_variable_size
```

Constructors

CONSTRUCTOR	DESCRIPTION
max_variable_size	Constructs an object of type <code>max_variable_size</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_variable_size::allocated

Increments the count of allocated memory blocks.

```
void allocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

This member function adds `_Nx` to the stored value `_Nallocs`. This member function is called after each successful call by `cache_freelist::allocate` to operator **new**. The argument `_Nx` is the number of memory blocks in the chunk allocated by operator **new**.

max_variable_size::deallocated

Decrements the count of allocated memory blocks.

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Nx</code>	The increment value.

Remarks

The member function subtracts `_Nx` from the stored value `_Nallocs`. This member function is called after each call by `cache_freelist::deallocate` to operator **delete**. The argument `_Nx` is the number of memory blocks in the chunk deallocated by operator **delete**.

max_variable_size::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
bool full();
```

Return Value

true if `_Nallocs / 16 + 16 <= _Nblocks`.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns **true**, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator **delete** to deallocate the block.

max_variable_size::max_variable_size

Constructs an object of type `max_variable_size`.

```
max_variable_size();
```

Remarks

The constructor initializes the stored values `_Nblocks` and `_Nallocs` to zero.

max_variable_size::released

Decrements the count of memory blocks on the free list.

```
void released();
```

Remarks

This member function decrements the stored value `_Nblocks`. The `released` member function of the current `max` class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

`max_variable_size::saved`

Increments the count of memory blocks on the free list.

```
void saved();
```

Remarks

This member function increments the stored value `_Nblocks`. This member function is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

[<allocators>](#)

rts_alloc Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The rts_alloc template class describes a [filter](#) that holds an array of cache instances and determines which instance to use for allocation and deallocation at runtime instead of at compile time.

Syntax

```
template <class Cache>
class rts_alloc
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The type of cache instances contained in the array. This can be cache_chunklist Class , cache_freelist , or cache_suballoc .

Remarks

This template class holds multiple block allocator instances and determines which instance to use for allocation or deallocation at runtime instead of at compile time. It is used with compilers that cannot compile rebind.

Member functions

MEMBER FUNCTION	DESCRIPTION
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

rts_alloc::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

The member function returns `caches[_IDX].allocate(count)`, where the index `_IDX` is determined by the requested block size *count*, or, if *count* is too large, it returns `operator new(count)`. `cache`, which represents the cache object.

rts_alloc::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

The member function calls `caches[_IDX].deallocate(ptr, count)`, where the index `_IDX` is determined by the requested block size *count*, or, if *count* is too large, it returns `operator delete(ptr)`.

rts_alloc::equals

Compares two caches for equality.

```
bool equals(const sync<_Cache>& _Other) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>_Cache</i>	The cache object associated with the filter.
<i>_Other</i>	The cache object to compare for equality.

Remarks

true if the result of `caches[0].equals(other.caches[0])`; otherwise, **false**. `caches` represents the array of cache objects.

See also

[ALLOCATOR_DECL](#)

<allocators>

sync_none Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [synchronization filter](#) that provides no synchronization.

Syntax

```
template <class Cache>
class sync_none
```

Parameters

PARAMETER	DESCRIPTION
<code>Cache</code>	The type of cache associated with the synchronization filter. This can be cache_chunklist , cache_freelist , or cache_suballoc .

Member functions

MEMBER FUNCTION	DESCRIPTION
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_none::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Remarks

The member function returns `cache.allocate(count)`, where `cache` is the cache object.

sync_none::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

The member function calls `cache.deallocate(ptr, count)`, where `cache` represents the cache object.

sync_none::equals

Compares two caches for equality.

```
bool equals(const sync<Cache>& Other) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The cache object of the synchronization filter.
<i>Other</i>	The cache object to compare for equality.

Return Value

The member function always returns **true**.

Remarks

See also

[<allocators>](#)

sync_per_container Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [synchronization filter](#) that provides a separate cache object for each allocator object.

Syntax

```
template <class Cache>
class sync_per_container
    : public Cache
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The type of cache associated with the synchronization filter. This can be cache_chunklist , cache_freelist , or cache_suballoc .

Member functions

MEMBER FUNCTION	DESCRIPTION
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_per_container::equals

Compares two caches for equality.

```
bool equals(const sync_per_container<Cache>& Other) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The cache object of the synchronization filter.
<i>Other</i>	The cache object to compare for equality.

Return Value

The member function always returns **false**.

Remarks

See also

[<allocators>](#)

sync_per_thread Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [synchronization filter](#) that provides a separate cache object for each thread.

Syntax

```
template <class Cache>
class sync_per_thread
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The type of cache associated with the synchronization filter. This can be cache_chunklist , cache_freelist , or cache_suballoc .

Remarks

Allocators that use `sync_per_thread` can compare equal even though blocks allocated in one thread cannot be deallocated from another thread. When using one of these allocators memory blocks allocated in one thread should not be made visible to other threads. In practice this means that a container that uses one of these allocators should only be accessed by a single thread.

Member functions

MEMBER FUNCTION	DESCRIPTION
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_per_thread::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Remarks

The member function returns the result of a call to `cache::allocate(count)` on the cache object belonging to the current thread. If no cache object has been allocated for the current thread, it first allocates one.

sync_per_thread::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

The member function calls `deallocate` on the cache object belonging to the current thread. If no cache object has been allocated for the current thread, it first allocates one.

sync_per_thread::equals

Compares two caches for equality.

```
bool equals(const sync<Cache>& Other) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The cache object of the synchronization filter.
<i>Other</i>	The cache object to compare for equality.

Return Value

false if no cache object has been allocated for this object or for *Other* in the current thread. Otherwise it returns the result of applying `operator==` to the two cache objects.

Remarks

See also

[<allocators>](#)

sync_shared Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a [synchronization filter](#) that uses a mutex to control access to a cache object that is shared by all allocators.

Syntax

```
template <class Cache>
class sync_shared
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The type of cache associated with the synchronization filter. This can be cache_chunklist , cache_freelist , or cache_suballoc .

Member functions

MEMBER FUNCTION	DESCRIPTION
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_shared::allocate

Allocates a block of memory.

```
void *allocate(std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>count</i>	The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

The member function locks the mutex, calls `cache.allocate(count)`, unlocks the mutex, and returns the result of the earlier call to `cache.allocate(count)`. `cache` represents the current cache object.

`sync_shared::deallocate`

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the first object to be deallocated from storage.
<i>count</i>	The number of objects to be deallocated from storage.

Remarks

This member function locks the mutex, calls `cache.deallocate(ptr, count)`, where `cache` represents the cache object, and then unlocks the mutex.

`sync_shared::equals`

Compares two caches for equality.

```
bool equals(const sync_shared<Cache>& Other) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>Cache</i>	The type of cache associated with the synchronization filter.
<i>Other</i>	The cache to compare for equality.

Return Value

true if the result of `cache.equals(Other.cache)`, where `cache` represents the cache object, is **true**; otherwise, **false**.

Remarks

See also

[<allocators>](#)

<array>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class **array** and several supporting templates.

Syntax

```
#include <array>
```

Classes

CLASS	DESCRIPTION
array	Stores a fixed-length sequence of elements.
tuple_element	Wraps the type of an array element.
tuple_size	Wraps the size of an array element.

Operators

OPERATOR	DESCRIPTION
operator==	array comparison, equal
operator!=	array comparison, not equal
operator<	array comparison, less than
operator>=	array comparison, greater than or equal
operator>	array comparison, greater than
operator<=	array comparison, less than or equal

Functions

FUNCTION	DESCRIPTION
get	Get specified array element.
swap	Exchanges the contents of one array with the contents of another array.

See also

[<tuple>](#)

[Header Files Reference](#)

<array> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

The <array> header includes two non-member functions, `get` and `swap`, that operate on **array** objects.

<code>get</code>	<code>swap</code>

get

Returns a reference to the specified element of the array.

```
template <int Index, class T, size_t N>
constexpr T& get(array<T, N>& arr) noexcept;

template <int Index, class T, size_t N>
constexpr const T& get(const array<T, N>& arr) noexcept;

template <int Index, class T, size_t N>
constexpr T&& get(array<T, N>&& arr) noexcept;
```

Parameters

Index

The element offset.

T

The type of an element.

N

The number of elements in the array.

arr

The array to select from.

Example

```

#include <array>
#include <iostream>

using namespace std;

typedef array<int, 4> MyArray;

int main()
{
    MyArray c0 { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& e : c0)
    {
        cout << " " << e;
    }
    cout << endl;

    // display odd elements " 1 3"
    cout << " " << get<1>(c0);
    cout << " " << get<3>(c0) << endl;
}

```

```

0 1 2 3
1 3

```

swap

A non-member template specialization of `std::swap` that swaps two **array** objects.

```

template <class Ty, std::size_t N>
void swap(array<Ty, N>& left, array<Ty, N>& right);

```

Parameters

Ty

The type of an element.

N

The size of the array.

left

The first array to swap.

right

The second array to swap.

Remarks

The template function executes `left.swap(right)`.

Example

```

// std__array__swap.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = { 4, 5, 6, 7 };
    c0.swap(c1);

    // display swapped contents " 4 5 6 7"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    swap(c0, c1);

    // display swapped contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
4 5 6 7
0 1 2 3

```

See also

[<array>](#)

<array> operators

11/9/2018 • 5 minutes to read • [Edit Online](#)

The <array> header includes these **array** non-member comparison template functions.

operator!=	operator>	operator>=
operator<	operator<=	operator==

operator!=

Array comparison, not equal.

```
template <Ty, std::size_t N>
bool operator!=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function returns `!(left == right)`.

Example

```

// std_array_operator_ne.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 != c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 != c1);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
4 5 6 7
false
true

```

operator<

Array comparison, less than.

```

template <Ty, std::size_t N>
bool operator<(
    const array<Ty, N>& left,
    const array<Ty, N>& right);

```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function overloads `operator<` to compare two objects of template class [array Class](#). The function returns `lexicographical_compare(left.begin(), left.end(), right.begin())`.

Example

```
// std_array_operator_lt.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 < c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 < c1);
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4 5 6 7
false
true
```

operator<=

Array comparison, less than or equal.

```
template <Ty, std::size_t N>
bool operator<=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function returns `!(right < left)`.

Example

```
// std__array__operator_le.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 <= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 <= c0);
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4 5 6 7
true
false
```

operator==

Array comparison, equal.

```
template <Ty, std::size_t N>
bool operator==(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function overloads `operator==` to compare two objects of template class [array Class](#). The function returns `equal(left.begin(), left.end(), right.begin())`.

Example

```
// std_array_operator_eq.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 == c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 == c1);
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4 5 6 7
true
false
```

operator>

Array comparison, greater than.

```
template <Ty, std::size_t N>
bool operator>(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function returns `(right < left)`.

Example

```
// std_array_operator_gt.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 > c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 > c0);
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4 5 6 7
false
true
```

operator>=

Array comparison, greater than or equal.

```
template <Ty, std::size_t N>
bool operator>=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right

Right container to compare.

Remarks

The template function returns `!(left < right)`.

Example

```
// std__array__operator_ge.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

    // display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 >= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 >= c1);
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4 5 6 7
true
false
```

See also

[`<array>`](#)

array Class (C++ Standard Library)

12/20/2018 • 21 minutes to read • [Edit Online](#)

Describes an object that controls a sequence of length `N` of elements of type `Ty`. The sequence is stored as an array of `Ty`, contained in the `array<Ty, N>` object.

Syntax

```
template <class Ty, std::size_t N>
class array;
```

Parameters

PARAMETER	DESCRIPTION
<code>Ty</code>	The type of an element.
<code>N</code>	The number of elements.

Members

TYPE DEFINITION	DESCRIPTION
<code>const_iterator</code>	The type of a constant iterator for the controlled sequence.
<code>const_pointer</code>	The type of a constant pointer to an element.
<code>const_reference</code>	The type of a constant reference to an element.
<code>const_reverse_iterator</code>	The type of a constant reverse iterator for the controlled sequence.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>reverse_iterator</code>	The type of a reverse iterator for the controlled sequence.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.

MEMBER FUNCTION	DESCRIPTION
<code>array</code>	Constructs an array object.
<code>assign</code>	Replaces all elements.
<code>at</code>	Accesses an element at a specified position.
<code>back</code>	Accesses the last element.
<code>begin</code>	Designates the beginning of the controlled sequence.
<code>cbegin</code>	Returns a random-access const iterator to the first element in the array.
<code>cend</code>	Returns a random-access const iterator that points just beyond the end of the array.
<code>crbegin</code>	Returns a const iterator to the first element in a reversed array.
<code>crend</code>	Returns a const iterator to the end of a reversed array.
<code>data</code>	Gets the address of the first element.
<code>empty</code>	Tests whether elements are present.
<code>end</code>	Designates the end of the controlled sequence.
<code>fill</code>	Replaces all elements with a specified value.
<code>front</code>	Accesses the first element.
<code>max_size</code>	Counts the number of elements.
<code>rbegin</code>	Designates the beginning of the reversed controlled sequence.
<code>rend</code>	Designates the end of the reversed controlled sequence.
<code>size</code>	Counts the number of elements.
<code>swap</code>	Swaps the contents of two containers.
OPERATOR	DESCRIPTION
<code>array::operator=</code>	Replaces the controlled sequence.
<code>array::operator[]</code>	Accesses an element at a specified position.

Remarks

The type has a default constructor `array()` and a default assignment operator `operator=`, and satisfies the

requirements for an `aggregate`. Therefore, objects of type `array<Ty, N>` can be initialized by using an aggregate initializer. For example,

```
array<int, 4> ai = { 1, 2, 3 };
```

creates the object `ai` that holds four integer values, initializes the first three elements to the values 1, 2, and 3, respectively, and initializes the fourth element to 0.

Requirements

Header: `<array>`

Namespace: `std`

`array::array`

Constructs an array object.

```
array();  
  
array(const array& right);
```

Parameters

right

Object or range to insert.

Remarks

The default constructor `array()` leaves the controlled sequence uninitialized (or default initialized). You use it to specify an uninitialized controlled sequence.

The copy constructor `array(const array& right)` initializes the controlled sequence with the sequence [*right*.begin(), *right*.end()). You use it to specify an initial controlled sequence that is a copy of the sequence controlled by the array object *right*.

Example

```

// std__array__array_array.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1(c0);

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3

```

array::assign

Obsolete in C++11, replaced by [fill](#). Replaces all elements.

```
void assign(const Ty& val);
```

Parameters

val

The value to assign.

Remarks

The member function replaces the sequence controlled by `*this` with a repetition of `N` elements of value *val*.

Example

```

// std_array_array_assign.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1;
    c1.assign(4);

    // display contents " 4 4 4 4"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
4 4 4 4

```

array::at

Accesses an element at a specified position.

```

reference at(size_type off);

constexpr const_reference at(size_type off) const;

```

Parameters

off

Position of element to access.

Remarks

The member functions return a reference to the element of the controlled sequence at position *off*. If that position is invalid, the function throws an object of class `out_of_range`.

Example

```

// std_array_array_at.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display odd elements " 1 3"
    std::cout << " " << c0.at(1);
    std::cout << " " << c0.at(3);
    std::cout << std::endl;

    return (0);
}

```

array::back

Accesses the last element.

```

reference back();

constexpr const_reference back() const;

```

Remarks

The member functions return a reference to the last element of the controlled sequence, which must be non-empty.

Example

```

// std_array_array_back.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display last element " 3"
    std::cout << " " << c0.back();
    std::cout << std::endl;

    return (0);
}

```

```
0 1 2 3
3
```

array::begin

Designates the beginning of the controlled sequence.

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Remarks

The member functions return a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Example

```
// std__array__array_begin.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::iterator it2 = c0.begin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
0
```

array::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const noexcept;
```

Return Value

A **const** random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin` , the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`array::cend`

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const noexcept;
```

Return Value

A random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

`array::const_iterator`

The type of a constant iterator for the controlled sequence.

```
typedef implementation-defined const_iterator;
```

Remarks

The type describes an object that can serve as a constant random-access iterator for the controlled sequence.

Example

```

// std_array_array_const_iterator.cpp
// compile with: /EHsc /W4
#include <array>
#include <iostream>

typedef std::array<int, 4> MyArray;

int main()
{
    MyArray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    std::cout << "it1:";
    for ( MyArray::const_iterator it1 = c0.begin();
          it1 != c0.end();
          ++it1 ) {
        std::cout << " " << *it1;
    }
    std::cout << std::endl;

    // display first element " 0"
    MyArray::const_iterator it2 = c0.begin();
    std::cout << "it2:";
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}

```

```

it1: 0 1 2 3
it2: 0

```

array::const_pointer

The type of a constant pointer to an element.

```

typedef const Ty *const_pointer;

```

Remarks

The type describes an object that can serve as a constant pointer to elements of the sequence.

Example

```

// std__array__array_const_pointer.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_pointer ptr = &c0.begin();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0

```

array::const_reference

The type of a constant reference to an element.

```

typedef const Ty& const_reference;

```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example


```

// std__array__array_const_reference.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_reference ref = *c0.begin();
    std::cout << " " << ref;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0

```

array::const_reverse_iterator

The type of a constant reverse iterator for the controlled sequence.

```

typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

```

Remarks

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

Example

```
// std_array_array_const_reverse_iterator.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display last element " 3"
    Myarray::const_reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
3
```

array::crbegin

Returns a const iterator to the first element in a reversed array.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse random-access iterator addressing the first element in a reversed array or addressing what had been the last element in the unreversed array.

Remarks

With the return value of `crbegin`, the array object cannot be modified.

Example

```
// array_crbegin.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

int main( )
{
    using namespace std;
    array<int, 2> v1 = {1, 2};
    array<int, 2>::iterator v1_Iter;
    array<int, 2>::const_reverse_iterator v1_rIter;

    v1_Iter = v1.begin( );
    cout << "The first element of array is "
         << *v1_Iter << "." << endl;

    v1_rIter = v1.crbegin( );
    cout << "The first element of the reversed array is "
         << *v1_rIter << "." << endl;
}
```

```
The first element of array is 1.
The first element of the reversed array is 2.
```

array::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed array.

```
const_reverse_iterator crend() const noexcept;
```

Return Value

A const reverse random-access iterator that addresses the location succeeding the last element in a reversed array (the location that had preceded the first element in the unreversed array).

Remarks

`crend` is used with a reversed array just as `array::cend` is used with a array.

With the return value of `crend` (suitably decremented), the array object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its array.

The value returned by `crend` should not be dereferenced.

Example

```
// array_crend.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

int main( )
{
    using namespace std;
    array<int, 2> v1 = {1, 2};
    array<int, 2>::const_reverse_iterator v1_rIter;

    for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
        cout << *v1_rIter << endl;
}
```

```
2
1
```

array::data

Gets the address of the first element.

```
Ty *data();

const Ty *data() const;
```

Remarks

The member functions return the address of the first element in the controlled sequence.

Example

```
// std__array__array_data.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::pointer ptr = c0.data();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
0
```

array::difference_type

The type of a signed distance between two elements.

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is a synonym for the type `std::ptrdiff_t`.

Example

```
// std_array_array_difference_type.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display distance first-last " -4"
    Myarray::difference_type diff = c0.begin() - c0.end();
    std::cout << " " << diff;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
-4
```

array::empty

Tests whether no elements are present.

```
constexpr bool empty() const;
```

Remarks

The member function returns true only if `N == 0`.

Example

```

// std__array__array_empty.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display whether c0 is empty " false"
    std::cout << std::boolalpha << " " << c0.empty();
    std::cout << std::endl;

    std::array<int, 0> c1;

    // display whether c1 is empty " true"
    std::cout << std::boolalpha << " " << c1.empty();
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
false
true

```

array::end

Designates the end of the controlled sequence.

```

reference end();

const_reference end() const;

```

Remarks

The member functions return a random-access iterator that points just beyond the end of the sequence.

Example

```
// std__array__array_end.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display last element " 3"
    Myarray::iterator it2 = c0.end();
    std::cout << " " << *--it2;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
3
```

array::fill

Erases a array and copies the specified elements to the empty array.

```
void fill(const Type& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of the element being inserted into the array.

Remarks

`fill` replaces each element of the array with the specified value.

Example

```

// array_fill.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

int main( )
{
    using namespace std;
    array<int, 2> v1 = {1, 2};
    array<int, 2>::iterator iter;

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v1.fill(3);
    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

```

array::front

Accesses the first element.

```

reference front();

constexpr const_reference front() const;

```

Remarks

The member functions return a reference to the first element of the controlled sequence, which must be non-empty.

Example

```

// std_array_array_front.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    std::cout << " " << c0.front();
    std::cout << std::endl;

    return (0);
}

```



```
0 1 2 3
0
```

array::iterator

The type of an iterator for the controlled sequence.

```
typedef implementation-defined iterator;
```

Remarks

The type describes an object that can serve as a random-access iterator for the controlled sequence.

Example

```
// std_array_array_iterator.cpp
// compile with: /EHsc /W4
#include <array>
#include <iostream>

typedef std::array<int, 4> MyArray;

int main()
{
    MyArray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    std::cout << "it1:";
    for ( MyArray::iterator it1 = c0.begin();
          it1 != c0.end();
          ++it1 ) {
        std::cout << " " << *it1;
    }
    std::cout << std::endl;

    // display first element " 0"
    MyArray::iterator it2 = c0.begin();
    std::cout << "it2:";
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

```
it1: 0 1 2 3

it2: 0
```

array::max_size

Counts the number of elements.

```
constexpr size_type max_size() const;
```

Remarks

The member function returns `N`.

Example

```
// std_array_array_max_size.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display (maximum) size " 4"
    std::cout << " " << c0.max_size();
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4
```

array::operator[]

Accesses an element at a specified position.

```
reference operator[](size_type off);

constexpr const_reference operator[](size_type off) const;
```

Parameters

off

Position of element to access.

Remarks

The member functions return a reference to the element of the controlled sequence at position *off*. If that position is invalid, the behavior is undefined.

There is also a non-member [get](#) function available to get a reference to an element of an **array**.

Example

```
// std__array__array_operator_sub.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display odd elements " 1 3"
    std::cout << " " << c0[1];
    std::cout << " " << c0[3];
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
1 3
```

array::operator=

Replaces the controlled sequence.

```
array<Value> operator=(array<Value> right);
```

Parameters

right

Container to copy.

Remarks

The member operator assigns each element of *right* to the corresponding element of the controlled sequence, then returns `*this`. You use it to replace the controlled sequence with a copy of the controlled sequence in *right*.

Example

```

// std__array__array_operator_as.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1;
    c1 = c0;

    // display copied contents " 0 1 2 3"
    for (Myarray::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3

```

array::pointer

The type of a pointer to an element.

```
typedef Ty *pointer;
```

Remarks

The type describes an object that can serve as a pointer to elements of the sequence.

Example

```

// std__array__array_pointer.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::pointer ptr = &c0.begin();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0

```

array::rbegin

Designates the beginning of the reversed controlled sequence.

```

reverse_iterator rbegin()noexcept;
const_reverse_iterator rbegin() const noexcept;

```

Remarks

The member functions return a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

Example

```

// std__array__array_rbegin.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display last element " 3"
    Myarray::const_reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
3

```

array::reference

The type of a reference to an element.

```

typedef Ty& reference;

```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

```

// std__array__array_reference.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::reference ref = *c0.begin();
    std::cout << " " << ref;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0

```

array::rend

Designates the end of the reversed controlled sequence.

```

reverse_iterator rend()noexcept;
const_reverse_iterator rend() const noexcept;

```

Remarks

The member functions return a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence)). Hence, it designates the end of the reverse sequence.

Example

```

// std__array__array_rend.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_reverse_iterator it2 = c0.rend();
    std::cout << " " << *--it2;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0

```

array::reverse_iterator

The type of a reverse iterator for the controlled sequence.

```

typedef std::reverse_iterator<iterator> reverse_iterator;

```

Remarks

The type describes an object that can serve as a reverse iterator for the controlled sequence.

Example


```

// std__array__array_reverse_iterator.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display last element " 3"
    Myarray::reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
3

```

array::size

Counts the number of elements.

```
constexpr size_type size() const;
```

Remarks

The member function returns `N`.

Example

```

// std__array__array_size.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display size " 4"
    std::cout << " " << c0.size();
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
4

```

array::size_type

The type of an unsigned distance between two element.

```

typedef std::size_t size_type;

```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is a synonym for the type `std::size_t`.

Example

```
// std_array_array_size_type.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display distance last-first " 4"
    Myarray::size_type diff = c0.end() - c0.begin();
    std::cout << " " << diff;
    std::cout << std::endl;

    return (0);
}
```

```
0 1 2 3
4
```

array::swap

Swaps the contents of this array with another array.

```
void swap(array& right);
```

Parameters

right

Array to swap contents with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. It performs a number of element assignments and constructor calls proportional to `N`.

There is also a non-member [swap](#) function available to swap two **array** instances.

Example

```

// std__array__array_swap.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};
    c0.swap(c1);

    // display swapped contents " 4 5 6 7"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    swap(c0, c1);

    // display swapped contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
4 5 6 7
0 1 2 3

```

array::value_type

The type of an element.

```
typedef Ty value_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

```

// std__array__array_value_type.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = {0, 1, 2, 3};

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
         it != c0.end(); ++it)
    {
        Myarray::value_type val = *it;
        std::cout << " " << val;
    }
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3

```

See also

[<array>](#)

<atomic>

10/31/2018 • 4 minutes to read • [Edit Online](#)

Defines classes and template classes to use to create types that support atomic operations.

Syntax

```
#include <atomic>
```

Remarks

NOTE

In code that's compiled by using **/clr**, this header is blocked.

An atomic operation has two key properties that help you use multiple threads to correctly manipulate an object without using mutex locks.

- Because an atomic operation is indivisible, a second atomic operation on the same object from a different thread can obtain the state of the object only before or after the first atomic operation.
- Based on its [memory_order](#) argument, an atomic operation establishes ordering requirements for the visibility of the effects of other atomic operations in the same thread. Consequently, it inhibits compiler optimizations that violate the ordering requirements.

On some platforms, it might not be possible to efficiently implement atomic operations for some types without using `mutex` locks. An atomic type is *lock-free* if no atomic operations on that type use locks.

C++11: In signal-handlers you can perform atomic operations on an object `obj` if `obj.is_lock_free()` or `atomic_is_lock_free(x)` are true.

The class [atomic_flag](#) provides a minimal atomic type that holds a **bool** flag. Its operations are always lock-free.

The template class `atomic<T>` stores an object of its argument type `T` and provides atomic access to that stored value. You can instantiate it by using any type that can be copied by using [memcpy](#) and tested for equality by using [memcmp](#). In particular, you can use it with user-defined types that meet these requirements and, in many cases, with floating-point types.

The template also has a set of specializations for integral types and a partial specialization for pointers. These specializations provide additional operations that are not available through the primary template.

Pointer Specializations

The `atomic<T*>` partial specializations apply to all pointer types. They provide methods for pointer arithmetic.

Integral Specializations

The `atomic<integral>` specializations apply to all integral types. They provide additional operations that are not available through the primary template.

Each `atomic<integral>` type has a corresponding macro that you can use in an `if directive` to determine at compile time whether operations on that type are lock-free. If the value of the macro is zero, operations on the type are not lock-free. If the value is 1, operations might be lock-free, and a runtime check is required. If the value is 2, operations are lock-free. You can use the function `atomic_is_lock_free` to determine at runtime whether operations on the type are lock-free.

For each of the integral types, there is a corresponding named atomic type that manages an object of that integral type. Each `atomic_integral` type has the same set of member functions as the corresponding instantiation of `atomic<T>` and can be passed to any of the non-member atomic functions.

<code>ATOMIC_INTEGRAL</code> TYPE	INTEGRAL TYPE	<code>ATOMIC_IS_LOCK_FREE</code> MACRO
<code>atomic_char</code>	char	<code>ATOMIC_CHAR_LOCK_FREE</code>
<code>atomic_schar</code>	signed char	<code>ATOMIC_CHAR_LOCK_FREE</code>
<code>atomic_uchar</code>	unsigned char	<code>ATOMIC_CHAR_LOCK_FREE</code>
<code>atomic_char16_t</code>	<code>char16_t</code>	<code>ATOMIC_CHAR16_T_LOCK_FREE</code>
<code>atomic_char32_t</code>	<code>char32_t</code>	<code>ATOMIC_CHAR32_T_LOCK_FREE</code>
<code>atomic_wchar_t</code>	wchar_t	<code>ATOMIC_WCHAR_T_LOCK_FREE</code>
<code>atomic_short</code>	short	<code>ATOMIC_SHORT_LOCK_FREE</code>
<code>atomic_ushort</code>	unsigned short	<code>ATOMIC_SHORT_LOCK_FREE</code>
<code>atomic_int</code>	int	<code>ATOMIC_INT_LOCK_FREE</code>
<code>atomic_uint</code>	unsigned int	<code>ATOMIC_INT_LOCK_FREE</code>
<code>atomic_long</code>	long	<code>ATOMIC_LONG_LOCK_FREE</code>
<code>atomic_ulong</code>	unsigned long	<code>ATOMIC_LONG_LOCK_FREE</code>
<code>atomic_llong</code>	long long	<code>ATOMIC_LLONG_LOCK_FREE</code>
<code>atomic_ullong</code>	unsigned long long	<code>ATOMIC_LLONG_LOCK_FREE</code>

Typedef names exist for specializations of the atomic template for some of the types that are defined in the header `<inttypes.h>`.

ATOMIC TYPE	TYPDEF NAME
<code>atomic_int8_t</code>	<code>atomic<int8_t></code>
<code>atomic_uint8_t</code>	<code>atomic<uint8_t></code>
<code>atomic_int16_t</code>	<code>atomic<int16_t></code>

ATOMIC TYPE	TYPEDDEF NAME
<code>atomic_uint16_t</code>	<code>atomic<uint16_t></code>
<code>atomic_int32_t</code>	<code>atomic<int32_t></code>
<code>atomic_uint32_t</code>	<code>atomic<uint32_t></code>
<code>atomic_int64_t</code>	<code>atomic<int64_t></code>
<code>atomic_uint64_t</code>	<code>atomic<uint64_t></code>
<code>atomic_int_least8_t</code>	<code>atomic<int_least8_t></code>
<code>atomic_uint_least8_t</code>	<code>atomic<uint_least8_t></code>
<code>atomic_int_least16_t</code>	<code>atomic<int_least16_t></code>
<code>atomic_uint_least16_t</code>	<code>atomic<uint_least16_t></code>
<code>atomic_int_least32_t</code>	<code>atomic<int_least32_t></code>
<code>atomic_uint_least32_t</code>	<code>atomic<uint_least32_t></code>
<code>atomic_int_least64_t</code>	<code>atomic<int_least64_t></code>
<code>atomic_uint_least64_t</code>	<code>atomic<uint_least64_t></code>
<code>atomic_int_fast8_t</code>	<code>atomic<int_fast8_t></code>
<code>atomic_uint_fast8_t</code>	<code>atomic<uint_fast8_t></code>
<code>atomic_int_fast16_t</code>	<code>atomic<int_fast16_t></code>
<code>atomic_uint_fast16_t</code>	<code>atomic<uint_fast16_t></code>
<code>atomic_int_fast32_t</code>	<code>atomic<int_fast32_t></code>
<code>atomic_uint_fast32_t</code>	<code>atomic<uint_fast32_t></code>
<code>atomic_int_fast64_t</code>	<code>atomic<int_fast64_t></code>
<code>atomic_uint_fast64_t</code>	<code>atomic<uint_fast64_t></code>
<code>atomic_intptr_t</code>	<code>atomic<intptr_t></code>
<code>atomic_uintptr_t</code>	<code>atomic<uintptr_t></code>
<code>atomic_size_t</code>	<code>atomic<size_t></code>

ATOMIC TYPE	TYPEDDEF NAME
<code>atomic_ptrdiff_t</code>	<code>atomic<ptrdiff_t></code>
<code>atomic_intmax_t</code>	<code>atomic<intmax_t></code>
<code>atomic_uintmax_t</code>	<code>atomic<uintmax_t></code>

Structs

NAME	DESCRIPTION
atomic Structure	Describes an object that performs atomic operations on a stored value.
atomic_flag Structure	Describes an object that atomically sets and clears a bool flag.

Enums

NAME	DESCRIPTION
memory_order Enum	Supplies symbolic names for synchronization operations on memory locations. These operations affect how assignments in one thread become visible in another.

Functions

In the following list, the functions that do not end in `_explicit` have the semantics of the corresponding `_explicit`, except that they have the implicit [memory_order](#) arguments of `memory_order_seq_cst`.

NAME	DESCRIPTION
atomic_compare_exchange_strong	Performs an <i>atomic compare and exchange</i> operation.
atomic_compare_exchange_strong_explicit	Performs an <i>atomic compare and exchange</i> operation.
atomic_compare_exchange_weak	Performs a <i>weak atomic compare and exchange</i> operation.
atomic_compare_exchange_weak_explicit	Performs a <i>weak atomic compare and exchange</i> operation.
atomic_exchange	Replaces a stored value.
atomic_exchange_explicit	Replaces a stored value.
atomic_fetch_add	Adds a specified value to an existing stored value.
atomic_fetch_add_explicit	Adds a specified value to an existing stored value.
atomic_fetch_and	Performs a bitwise <code>and</code> on a specified value and an existing stored value.

NAME	DESCRIPTION
atomic_fetch_and_explicit	Performs a bitwise <code>and</code> on a specified value and an existing stored value.
atomic_fetch_or	Performs a bitwise <code>or</code> on a specified value and an existing stored value.
atomic_fetch_or_explicit	Performs a bitwise <code>or</code> on a specified value and an existing stored value.
atomic_fetch_sub	Subtracts a specified value from an existing stored value.
atomic_fetch_sub_explicit	Subtracts a specified value from an existing stored value.
atomic_fetch_xor	Performs a bitwise <code>exclusive or</code> on a specified value and an existing stored value.
atomic_fetch_xor_explicit	Performs a bitwise <code>exclusive or</code> on a specified value and an existing stored value.
atomic_flag_clear	Sets the flag in an <code>atomic_flag</code> object to false .
atomic_flag_clear_explicit	Sets the flag in an <code>atomic_flag</code> object to false .
atomic_flag_test_and_set	Sets the flag in an <code>atomic_flag</code> object to true .
atomic_flag_test_and_set_explicit	Sets the flag in an <code>atomic_flag</code> object to true .
atomic_init	Sets the stored value in an <code>atomic</code> object.
atomic_is_lock_free	Specifies whether atomic operations on a specified object are lock-free.
atomic_load	Atomically retrieves a value.
atomic_load_explicit	Atomically retrieves a value.
atomic_signal_fence	Acts as a <i>fence</i> that establishes memory ordering requirements between fences in a calling thread that has signal handlers executed in the same thread.
atomic_store	Atomically stores a value.
atomic_store_explicit	Atomically stores a value.
atomic_thread_fence	Acts as a <i>fence</i> that establishes memory ordering requirements with respect to other fences.
kill_dependency	Breaks a possible dependency chain.

See also

[Header Files Reference](#)

[C++ Standard Library Reference](#)

atomic Structure

10/31/2018 • 11 minutes to read • [Edit Online](#)

Describes an object that performs atomic operations on a stored value of type *Ty*.

Syntax

```
template <class Ty>
struct atomic;
```

Members

MEMBER	DESCRIPTION
Constructor	
atomic	Constructs an atomic object.
Operators	
atomic::operator Ty	Reads and returns the stored value. (atomic::load)
atomic::operator=	Uses a specified value to replace the stored value. (atomic::store)
atomic::operator++	Increments the stored value. Used only by integral and pointer specializations.
atomic::operator+=	Adds a specified value to the stored value. Used only by integral and pointer specializations.
atomic::operator--	Decrements the stored value. Used only by integral and pointer specializations.
atomic::operator-=	Subtracts a specified value from the stored value. Used only by integral and pointer specializations.
atomic::operator&=	Performs a bitwise and on a specified value and the stored value. Used only by integral specializations.
atomic::operator =	Performs a bitwise or on a specified value and the stored value. Used only by integral specializations.
atomic::operator^=	Performs a bitwise exclusive or on a specified value and the stored value. Used only by integral specializations.
Functions	

MEMBER	DESCRIPTION
<code>compare_exchange_strong</code>	Performs an <i>atomic_compare_and_exchange</i> operation on this and returns the result.
<code>compare_exchange_weak</code>	Performs a <i>weak_atomic_compare_and_exchange</i> operation on this and returns the result.
<code>fetch_add</code>	Adds a specified value to the stored value.
<code>fetch_and</code>	Performs a bitwise and on a specified value and the stored value.
<code>fetch_or</code>	Performs a bitwise or on a specified value and the stored value.
<code>fetch_sub</code>	Subtracts a specified value from the stored value.
<code>fetch_xor</code>	Performs a bitwise exclusive or on a specified value and the stored value.
<code>is_lock_free</code>	Specifies whether atomic operations on this are <i>lock free</i> . An atomic type is <i>lock free</i> if no atomic operations on that type use locks.
<code>load</code>	Reads and returns the stored value.
<code>store</code>	Uses a specified value to replace the stored value.

Remarks

The type *Ty* must be *trivially copyable*. That is, using `memcpy` to copy its bytes must produce a valid *Ty* object that compares equal to the original object. The `compare_exchange_weak` and `compare_exchange_strong` member functions use `memcpy` to determine whether two *Ty* values are equal. These functions will not use a *Ty*-defined `operator==`. The member functions of `atomic` use `memcpy` to copy values of type *Ty*.

A partial specialization, **`atomic<Ty*>`**, exists for all pointer types. The specialization enables the addition of an offset to the managed pointer value or the subtraction of an offset from it. The arithmetic operations take an argument of type `ptrdiff_t` and adjust that argument according to the size of *Ty* to be consistent with ordinary address arithmetic.

A specialization exists for every integral type except **`bool`**. Each specialization provides a rich set of methods for atomic arithmetic and logical operations.

<code>atomic<char></code>	<code>atomic<signed char></code>	<code>atomic<unsigned char></code>
<code>atomic<char16_t></code>	<code>atomic<char32_t></code>	<code>atomic<wchar_t></code>
<code>atomic<short></code>	<code>atomic<unsigned short></code>	<code>atomic<int></code>
<code>atomic<unsigned int></code>	<code>atomic<long></code>	<code>atomic<unsigned long></code>

atomic<long long>	atomic<unsigned long long>	

Integral specializations are derived from corresponding `atomic_integral` types. For example, **atomic<unsigned int>** is derived from `atomic_uint`.

Requirements

Header: `<atomic>`

Namespace: `std`

atomic::atomic

Constructs an atomic object.

```
atomic();
atomic( const atomic& );
atomic( Ty Value ) noexcept;
```

Parameters

Value

Initialization value.

Remarks

Atomic objects cannot be copied or moved.

Objects that are instantiations of `atomic<Ty>` can be initialized only by the constructor that takes an argument of type *Ty* and not by using aggregate initialization. However, `atomic_integral` objects can be initialized only by using aggregate initialization.

```
atomic<int> ai0 = ATOMIC_VAR_INIT(0);
atomic<int> ai1(0);
```

atomic::operator Ty

The operator for the type specified to the template, `atomic<Ty>`. Retrieves the stored value in ***this**.

```
atomic<Ty>::operator Ty() const volatile noexcept;
atomic<Ty>::operator Ty() const noexcept;
```

Remarks

This operator applies the `memory_order_seq_cst` [memory_order](#).

atomic::operator=

Stores a specified value.

```
Ty operator=(
    Ty Value
) volatile noexcept;
Ty operator=(
    Ty Value
) noexcept;
```

Parameters

Value

A *Ty* object.

Return Value

Returns *Value*.

atomic::operator++

Increments the stored value. Used only by integral and pointer specializations.

```
Ty atomic<Ty>::operator++(int) volatile noexcept;
Ty atomic<Ty>::operator++(int) noexcept;
Ty atomic<Ty>::operator++() volatile noexcept;
Ty atomic<Ty>::operator++() noexcept;
```

Return Value

The first two operators return the incremented value; the last two operators return the value before the increment.

The operators use the `memory_order_seq_cst` [memory_order](#).

atomic::operator+=

Adds a specified value to the stored value. Used only by integral and pointer specializations.

```
Ty atomic<Ty>::operator+=(
    Ty Value
) volatile noexcept;
Ty atomic<Ty>::operator+=(
    Ty Value
) noexcept;
```

Parameters

Value

An integral or pointer value.

Return Value

A *Ty* object that contains the result of the addition.

Remarks

This operator uses the `memory_order_seq_cst` [memory_order](#).

atomic::operator--

Decrements the stored value. Used only by integral and pointer specializations.

```
Ty atomic<Ty>::operator--(int) volatile noexcept;
Ty atomic<Ty>::operator--(int) noexcept;
Ty atomic<Ty>::operator--() volatile noexcept;
Ty atomic<Ty>::operator--() noexcept;
```

Return Value

The first two operators return the decremented value; the last two operators return the value before the decrement. The operators use the `memory_order_seq_cst` [memory_order](#).

atomic::operator-=

Subtracts a specified value from the stored value. Used only by integral and pointer specializations.

```
Ty atomic<Ty>::operator-=(
    Ty Value
) volatile noexcept;
Ty atomic<Ty>::operator-=(
    Ty Value
) noexcept;
```

Parameters

Value

An integral or pointer value.

Return Value

A *Ty* object that contains the result of the subtraction.

Remarks

This operator uses the `memory_order_seq_cst` [memory_order](#).

atomic::operator&=

Performs a bitwise and on a specified value and the stored value of ***this**. Used only by integral specializations.

```
atomic<Ty>::operator&= (
    Ty Value
) volatile noexcept;
atomic<Ty>::operator&= (
    Ty Value
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Return Value

The result of the bitwise and.

Remarks

This operator performs a read-modify-write operation to replace the stored value of ***this** with a bitwise and of *Value* and the current value that is stored in ***this**, within the constraints of the `memory_order_seq_cst` [memory_order](#).

atomic::operator|=

Performs a bitwise or on a specified value and the stored value of ***this**. Used only by integral specializations.

```
atomic<Ty>::operator|= (
    Ty Value
) volatile noexcept;
atomic<Ty>::operator|= (
    Ty Value
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Return Value

The result of the bitwise or.

Remarks

This operator performs a read-modify-write operation to replace the stored value of ***this** with a bitwise or of *Value* and the current value that is stored in ***this**, within the constraints of the `memory_order_seq_cst` [memory_order](#) constraints.

atomic::operator^=

Performs a bitwise exclusive or on a specified value and the stored value of ***this**. Used only by integral specializations.

```
atomic<Ty>::operator^= (
    Ty Value
) volatile noexcept;
atomic<Ty>::operator^= (
    Ty Value
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Return Value

The result of the bitwise exclusive or.

Remarks

This operator performs a read-modify-write operation to replace the stored value of ***this** with a bitwise exclusive or of *Value* and the current value that is stored in ***this**, within the constraints of the `memory_order_seq_cst` [memory_order](#) constraints.

atomic::compare_exchange_strong

Performs an atomic compare and exchange operation on ***this**.

```

bool compare_exchange_strong(
    Ty& Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2
) volatile noexcept;
bool compare_exchange_strong(
    Ty& Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2
) noexcept;
bool compare_exchange_strong(
    Ty& Exp,
    Ty Value,
    memory_order Order1 = memory_order_seq_cst
) volatile noexcept;
bool compare_exchange_strong(
    Ty& Exp,
    Ty Value,
    memory_order Order1 = memory_order_seq_cst
) noexcept;

```

Parameters

Exp

A value of type *Ty*.

Value

A value of type *Ty*.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument.

Return Value

A **bool** that indicates the result of the value comparison.

Remarks

This atomic compare and exchange operation compares the value that is stored in ***this** with *Exp*. If the values are equal, the operation replaces the value that is stored in ***this** with *Value* by using a read-modify-write operation and applying the memory order constraints that are specified by *Order1*. If the values are not equal, the operation uses the value that is stored in ***this** to replace *Exp* and applies the memory order constraints that are specified by *Order2*.

Overloads that do not have a second `memory_order` use an implicit *Order2* that is based on the value of *Order1*. If *Order1* is `memory_order_acq_rel`, *Order2* is `memory_order_acquire`. If *Order1* is `memory_order_release`, *Order2* is `memory_order_relaxed`. In all other cases, *Order2* is equal to *Order1*.

For overloads that take two `memory_order` parameters, the value of *Order2* must not be `memory_order_release` or `memory_order_acq_rel`, and must not be stronger than the value of *Order1*.

atomic::compare_exchange_weak

Performs a weak atomic compare and exchange operation on ***this**.

```

bool compare_exchange_weak(
    Ty& Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2
) volatile noexcept;
bool compare_exchange_weak(
    Ty& Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2
) noexcept;
bool compare_exchange_weak(
    Ty& Exp,
    Ty Value,
    memory_order Order1 = memory_order_seq_cst
) volatile noexcept;
bool compare_exchange_weak(
    Ty& Exp,
    Ty Value,
    memory_order Order1 = memory_order_seq_cst
) noexcept;

```

Parameters

Exp

A value of type *Ty*.

Value

A value of type *Ty*.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument.

Return Value

A **bool** that indicates the result of the value comparison.

Remarks

This atomic compare and exchange operation compares the value that is stored in ***this** with *Exp*. If the values are equal, the operation replaces the value that is stored in ***this** with *Value* by using a read-modify-write operation and applying the memory order constraints that are specified by *Order1*. If the values are not equal, the operation uses the value that is stored in ***this** to replace *Exp* and applies the memory order constraints that are specified by *Order2*.

A weak atomic compare and exchange operation performs an exchange if the compared values are equal. If the values are not equal, the operation is not guaranteed to perform an exchange.

Overloads that do not have a second `memory_order` use an implicit *Order2* that is based on the value of *Order1*. If *Order1* is `memory_order_acq_rel`, *Order2* is `memory_order_acquire`. If *Order1* is `memory_order_release`, *Order2* is `memory_order_relaxed`. In all other cases, *Order2* is equal to *Order1*.

For overloads that take two `memory_order` parameters, the value of *Order2* must not be `memory_order_release` or `memory_order_acq_rel`, and must not be stronger than the value of *Order1*.

atomic::exchange

Uses a specified value to replace the stored value of ***this**.

```
Ty atomic<Ty>::exchange(  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) volatile noexcept;  
Ty atomic<Ty>::exchange(  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

The stored value of ***this** before the exchange.

Remarks

This operation performs a read-modify-write operation to use *Value* to replace the value that is stored in ***this**, within the memory constraints that are specified by *Order*.

atomic::fetch_add

Fetches the value stored in ***this**, and then adds a specified value to the stored value.

```
Ty atomic<Ty>::fetch_add (  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) volatile noexcept;  
Ty atomic<Ty>::fetch_add (  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the value stored in ***this** prior to the addition.

Remarks

The `fetch_add` method performs a read-modify-write operation to atomically add *Value* to the stored value in ***this**, and applies the memory constraints that are specified by *Order*.

atomic::fetch_and

Performs a bitwise and on a value and an existing value that is stored in ***this**.

```

Ty atomic<Ty>::fetch_and (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_and (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) noexcept;

```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the bitwise and.

Remarks

The `fetch_and` method performs a read-modify-write operation to replace the stored value of ***this** with a bitwise and of *Value* and the current value that is stored in ***this**, within the memory constraints that are specified by *Order*.

atomic::fetch_or

Performs a bitwise or on a value and an existing value that is stored in ***this**.

```

Ty atomic<Ty>::fetch_or (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_or (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) noexcept;

```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the bitwise or.

Remarks

The `fetch_or` method performs a read-modify-write operation to replace the stored value of ***this** with a bitwise or of *Value* and the current value that is stored in ***this**, within the memory constraints that are specified by *Order*.

atomic::fetch_sub

Subtracts a specified value from the stored value.

```

Ty atomic<Ty>::fetch_sub (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_sub (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) noexcept;

```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the subtraction.

Remarks

The `fetch_sub` method performs a read-modify-write operation to atomically subtract *Value* from the stored value in ***this**, within the memory constraints that are specified by *Order*.

atomic::fetch_xor

Performs a bitwise exclusive or on a value and an existing value that is stored in ***this**.

```

Ty atomic<Ty>::fetch_xor (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_xor (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) noexcept;

```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the bitwise exclusive or.

Remarks

The `fetch_xor` method performs a read-modify-write operation to replace the stored value of ***this** with a bitwise exclusive or of *Value* and the current value that is stored in ***this**, and applies the memory constraints that are specified by *Order*.

atomic::is_lock_free

Specifies whether atomic operations on ***this** are lock free.

```
bool is_lock_free() const volatile noexcept;
```

Return Value

true if atomic operations on ***this** are lock free; otherwise, false.

Remarks

An atomic type is lock free if no atomic operations on that type use locks.

atomic::load

Retrieves the stored value in ***this**, within the specified memory constraints.

```
Ty atomic::load(  
    memory_order Order = memory_order_seq_cst  
) const volatile noexcept;  
Ty atomic::load(  
    memory_order Order = memory_order_seq_cst  
) const noexcept;
```

Parameters

Order

A `memory_order`. *Order* must not be `memory_order_release` or `memory_order_acq_rel`.

Return Value

The retrieved value that is stored in ***this**.

atomic::store

Stores a specified value.

```
void atomic<Ty>::store(  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) volatile noexcept;  
void atomic<Ty>::store(  
    Ty Value,  
    memory_order Order = memory_order_seq_cst  
) noexcept;
```

Parameters

Value

A *Ty* object.

Order

A `memory_order` constraint.

Remarks

This member function atomically stores *Value* in `*this`, within the memory constraints that are specified by *Order*.

See also

[<atomic>](#)

[Header Files Reference](#)

atomic_flag Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that atomically sets and clears a **bool** flag. Operations on atomic flags are always lock-free.

Syntax

```
struct atomic_flag;
```

Members

Public Methods

NAME	DESCRIPTION
clear	Sets the stored flag to false .
test_and_set	Sets the stored flag to true and returns the initial flag value.

Remarks

`atomic_flag` objects can be passed to the non-member functions [atomic_flag_clear](#), [atomic_flag_clear_explicit](#), [atomic_flag_test_and_set](#), and [atomic_flag_test_and_set_explicit](#). They can be initialized by using the value `ATOMIC_FLAG_INIT`.

Requirements

Header: <atomic>

Namespace: std

atomic_flag::clear

Sets the **bool** flag that is stored in `*this` to **false**, within the specified [memory_order](#) constraints.

```
void atomic_flag::clear(memory_order Order = memory_order_seq_cst) volatile noexcept;  
void atomic_flag::clear(memory_order Order = memory_order_seq_cst) noexcept;
```

Parameters

Order

A [memory_order](#).

atomic_flag::test_and_set

Sets the **bool** flag that is stored in `*this` to **true**, within the specified [memory_order](#) constraints.

```
bool atomic_flag::test_and_set(memory_order Order = memory_order_seq_cst) volatile noexcept;  
bool atomic_flag::test_and_set(memory_order Order = memory_order_seq_cst) noexcept;
```

Parameters

Order

A [memory_order](#).

Return Value

The initial value of the flag that is stored in `*this`.

See also

[<atomic>](#)

<atomic> functions

10/31/2018 • 15 minutes to read • [Edit Online](#)

atomic_compare_exchange_strong	atomic_compare_exchange_strong_explicit	atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit	atomic_exchange	atomic_exchange_explicit
atomic_fetch_add	atomic_fetch_add_explicit	atomic_fetch_and
atomic_fetch_and_explicit	atomic_fetch_or	atomic_fetch_or_explicit
atomic_fetch_sub	atomic_fetch_sub_explicit	atomic_fetch_xor
atomic_fetch_xor_explicit	atomic_flag_clear	atomic_flag_clear_explicit
atomic_flag_test_and_set	atomic_flag_test_and_set_explicit	atomic_init
atomic_is_lock_free	atomic_load	atomic_load_explicit
atomic_signal_fence	atomic_store	atomic_store_explicit
atomic_thread_fence	kill_dependency	

atomic_compare_exchange_strong

Performs an atomic compare and exchange operation.

```
template <class Ty>
inline bool atomic_compare_exchange_strong(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Value) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;
```

Parameters

Atom

A pointer to an *atomic* object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Return Value

true if the values are equal, otherwise **false**.

Remarks

This method performs an atomic compare and exchange operation by using implicit `memory_order_seq_cst` `memory_order` arguments. For more information, see [atomic_compare_exchange_strong_explicit](#).

atomic_compare_exchange_strong_explicit

Performs an *atomic compare and exchange* operation.

```
template <class T>
inline bool atomic_compare_exchange_strong_explicit(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong_explicit(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument. The value of *Order2* cannot be `memory_order_release` or `memory_order_acq_rel`, it cannot be stronger than the value of *Order1*.

Return Value

true if the values are equal, otherwise **false**.

Remarks

An *atomic compare and exchange operation* compares the value that is stored in the object that is pointed to by *Atom* against the value that is pointed to by *Exp*. If the values are equal, the value that is stored in the object that is pointed to by *atom* is replaced with *Value* by using a `read-modify-write` operation and applying the memory order constraints that are specified by *Order1*. If the values are not equal, the operation replaces the value that is pointed to by *Exp* with the value that is stored in the object that is pointed to by *Atom* and applies the memory order constraints that are specified by *Order2*.

atomic_compare_exchange_weak

Performs a *weak atomic compare and exchange* operation.

```
template <class Ty>
inline bool atomic_compare_exchange_strong(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Return Value

true if the values are equal, otherwise **false**.

Remarks

This method performs a *weak atomic compare and exchange operation* that has implicit `memory_order_seq_cst` [memory_order](#) arguments. For more information, see [atomic_compare_exchange_weak_explicit](#).

atomic_compare_exchange_weak_explicit

Performs a *weak atomic compare and exchange* operation.

```
template <class Ty>
inline bool atomic_compare_exchange_weak_explicit(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_weak_explicit(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument. The value of *Order2* cannot be `memory_order_release` or `memory_order_acq_rel`, nor can it be stronger than the value of *Order1*.

Return Value

true if the values are equal, otherwise **false**.

Remarks

Both the strong and weak flavors of an *atomic compare and exchange operation* guarantee that they do not store the new value if the expected and current values are not equal. The strong flavor guarantees that it will store the new value if the expected and current values are equal. The weak flavor may sometimes return **false** and not store the new value even if the current and expected values are equal. In other words, the function will return **false**, but a later examination of the expected value might reveal that it did not change, and therefore should have compared as equal.

atomic_exchange

Uses *Value* to replace the stored value of *Atom*.

```
template <class T>
inline Ty atomic_exchange(volatile atomic<Ty>* _Atom, Ty Value) noexcept;

template <class Ty>
inline T atomic_exchange(atomic<Ty>* Atom, Ty Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Value

A value of type `Ty`.

Return Value

The stored value of *Atom* before the exchange.

Remarks

The `atomic_exchange` function performs a `read-modify-write` operation to exchange the value that is stored in *Atom* with *Value*, using the `memory_order_seq_cst` `memory_order`.

atomic_exchange_explicit

Replaces the stored value of *Atom* with *Value*.

```
template <class Ty>
inline Ty atomic_exchange_explicit(
    volatile atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;

template <class Ty>
inline Ty atomic_exchange_explicit(
    atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Value

A value of type `Ty`.

Order

A [memory_order](#).

Return Value

The stored value of *Atom* before the exchange.

Remarks

The `atomic_exchange_explicit` function performs a `read-modify-write` operation to exchange the value that is stored in *Atom* with *Value*, within the memory constraints that are specified by *Order*.

atomic_fetch_add

Adds a value to an existing value that is stored in an `atomic` object.

```
template <class T>
T* atomic_fetch_add(volatile atomic<T*>* Atom, ptrdiff_t Value) noexcept;
template <class T>
T* atomic_fetch_add(atomic<T*>* Atom, ptrdiff_t Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_add` function performs a `read-modify-write` operation to atomically add *Value* to the stored value in *Atom*, using the `memory_order_seq_cst` [memory_order](#) constraint.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
integral atomic_fetch_add(volatile atomic-integral* Atom, integral Value) noexcept;
```

```
integral atomic_fetch_add(atomic-integral* Atom, integral Value) noexcept;
```

atomic_fetch_add_explicit

Adds a value to an existing value that is stored in an `atomic` object.

```
template <class T>
T* atomic_fetch_add_explicit(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;
```

```
template <class T>
T* atomic_fetch_add_explicit(
    atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_add_explicit` function performs a `read-modify-write` operation to atomically add *Value* to the stored value in *Atom*, within the `memory_order` constraints that are specified by `Order`.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
integral atomic_fetch_add_explicit(
    volatile atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;
```

```
integral atomic_fetch_add_explicit(
    atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;
```

atomic_fetch_and

Performs a bitwise `and` on a value and an existing value that is stored in an `atomic` object.


```
template <class T>
inline T atomic_fetch_and(volatile atomic<T>* Atom, T Value) noexcept;
template <class T>
inline T atomic_fetch_and(volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_and` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `and` of *Value* and the current value that is stored in *Atom*, using the `memory_order_seq_cst` [memory_order](#) constraint.

atomic_fetch_and_explicit

Performs a bitwise `and` of a value and an existing value that is stored in an `atomic` object.

```
template <class T>
inline T atomic_fetch_and_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_and_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A [memory_order](#).

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_and_explicit` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `and` of *Value* and the current value that is stored in *Atom*, within the memory constraints that are specified by *Order*.

atomic_fetch_or

Performs a bitwise `or` on a value and an existing value that is stored in an `atomic` object.

```
template <class T>
inline T atomic_fetch_or (volatile atomic<T>* Atom, T Value) noexcept;
template <class T>
inline T atomic_fetch_or (volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_or` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `or` of *Value* and the current value that is stored in *Atom*, using the `memory_order_seq_cst` [memory_order](#).

atomic_fetch_or_explicit

Performs a bitwise `or` on a value and an existing value that is stored in an `atomic` object.

```
template <class T>
inline T atomic_fetch_or_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_or_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A [memory_order](#).

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_or_explicit` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `or` of *Value* and the current value that is stored in *Atom*, within the [memory_order](#) constraints specified by *Order*.

atomic_fetch_sub

Subtracts a value from an existing value that is stored in an `atomic` object.

```
template <class T>
T* atomic_fetch_sub(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value) noexcept;

template <class T>
T* atomic_fetch_sub(
    atomic<T*>* Atom,
    ptrdiff_t Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_sub` function performs a `read-modify-write` operation to atomically subtract *Value* from the stored value in *Atom*, using the `memory_order_seq_cst` [memory_order](#) constraint.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
integral atomic_fetch_sub(volatile atomic-integral* Atom, integral Value) noexcept;
integral atomic_fetch_sub(atomic-integral* Atom, integral Value) noexcept;
```

atomic_fetch_sub_explicit

Subtracts a value from an existing value that is stored in an `atomic` object.

```
template <class T>
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;

template <class T>
T* atomic_fetch_sub_explicit(
    atomic<T*>* Atom,
    ptrdiff_t Value, memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_sub_explicit` function performs a `read-modify-write` operation to atomically subtract *Value* from the stored value in *Atom*, within the `memory_order` constraints that are specified by `Order`.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
integral atomic_fetch_sub_explicit(
    volatile atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;

integral atomic_fetch_sub_explicit(
    atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;
```

atomic_fetch_xor

Performs a bitwise `exclusive or` on a value and an existing value that is stored in an `atomic` object.

```
template <class T>
inline T atomic_fetch_xor(volatile atomic<T>* Atom, T Value) noexcept;

template <class T>
inline T atomic_fetch_xor(volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_xor` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `exclusive or` of *Value* and the current value that is stored in *Atom*, using the `memory_order_seq_cst` `memory_order`.

atomic_fetch_xor_explicit

Performs a bitwise `exclusive or` on a value and an existing value that is stored in an `atomic` object.

```

template <class T>
inline T atomic_fetch_xor_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_xor_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A [memory_order](#).

Return Value

The value contained by the atomic object immediately before the operation was performed.

Remarks

The `atomic_fetch_xor_explicit` function performs a `read-modify-write` operation to replace the stored value of *Atom* with a bitwise `exclusive or` of *Value* and the current value that is stored in *Atom*, within the [memory_order](#) constraints that are specified by *Order*.

atomic_flag_clear

Sets the **bool** flag in an [atomic_flag](#) object to **false**, within the `memory_order_seq_cst` [memory_order](#).

```

inline void atomic_flag_clear(volatile atomic_flag* Flag) noexcept;
inline void atomic_flag_clear(atomic_flag* Flag) noexcept;

```

Parameters

Flag

A pointer to an `atomic_flag` object.

atomic_flag_clear_explicit

Sets the **bool** flag in an [atomic_flag](#) object to **false**, within the specified [memory_order](#) constraints.

```

inline void atomic_flag_clear_explicit(volatile atomic_flag* Flag, memory_order Order) noexcept;
inline void atomic_flag_clear_explicit(atomic_flag* Flag, memory_order Order) noexcept;

```

Parameters

Flag

A pointer to an `atomic_flag` object.

Order

A [memory_order](#).

atomic_flag_test_and_set

Sets the **bool** flag in an [atomic_flag](#) object to **true**, within the constraints of the `memory_order_seq_cst` [memory_order](#).

```
inline bool atomic_flag_test_and_set(volatile atomic_flag* Flag,) noexcept;
inline bool atomic_flag_test_and_set(atomic_flag* Flag,) noexcept;
```

Parameters

Flag

A pointer to an `atomic_flag` object.

Return Value

The initial value of *Flag*.

atomic_flag_test_and_set_explicit

Sets the **bool** flag in an [atomic_flag](#) object to **true**, within the specified [memory_order](#) constraints.

```
inline bool atomic_flag_test_and_set_explicit(volatile atomic_flag* Flag, memory_order Order) noexcept;
inline bool atomic_flag_test_and_set_explicit(atomic_flag* Flag, memory_order Order) noexcept;
```

Parameters

Flag

A pointer to an `atomic_flag` object.

Order

A [memory_order](#).

Return Value

The initial value of *Flag*.

atomic_init

Sets the stored value in an `atomic` object.

```
template <class Ty>
inline void atomic_init(volatile atomic<Ty>* Atom, Ty Value) noexcept;
template <class Ty>
inline void atomic_init(atomic<Ty>* Atom, Ty Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Value

A value of type `Ty`.

Remarks

`atomic_init` is not an atomic operation. It is not thread-safe.

atomic_is_lock_free

Specifies whether atomic operations on an `atomic` object are *lock-free*.

```
template <class T>
inline bool atomic_is_lock_free(const volatile atomic<T>* Atom) noexcept;
template <class T>
inline bool atomic_is_lock_free(const atomic<T>* Atom) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Return Value

true if atomic operations on *Atom* are lock-free; otherwise, **false**.

Remarks

An atomic type is lock-free if no atomic operations on that type use locks. If this function returns true, the type is safe to use in signal-handlers.

atomic_load

Retrieves the stored value in an `atomic` object.

```
template <class Ty>
inline Ty atomic_load(const volatile atomic<Ty>* Atom) noexcept;
template <class Ty>
inline Ty atomic_load(const atomic<Ty>* Atom) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Return Value

The retrieved value that is stored in *Atom*.

Remarks

`atomic_load` implicitly uses the `memory_order_seq_cst` [memory_order](#).

atomic_load_explicit

Retrieves the stored value in an `atomic` object, within a specified [memory_order](#).

```
template <class Ty>
inline Ty atomic_load_explicit(const volatile atomic<Ty>* Atom, memory_order Order) noexcept;
template <class Ty>
inline Ty atomic_load_explicit(const atomic<Ty>* Atom, memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Order

A [memory_order](#). Do not use `memory_order_release` or `memory_order_acq_rel`.

Return Value

The retrieved value that is stored in *Atom*.

atomic_signal_fence

Acts as a *fence*—which is a memory synchronization primitive that enforces ordering between load/store operations—between other fences in a calling thread that have signal handlers that are executed in the same thread.

```
inline void atomic_signal_fence(memory_order Order) noexcept;
```

Parameters

Order

A memory ordering constraint that determines fence type.

Remarks

The *Order* argument determines fence type.

<code>memory_order_relaxed</code>	The fence has no effect.
<code>memory_order_consume</code>	The fence is an acquire fence.
<code>memory_order_acquire</code>	The fence is an acquire fence.
<code>memory_order_release</code>	The fence is a release fence.
<code>memory_order_acq_rel</code>	The fence is both an acquire fence and a release fence.
<code>memory_order_seq_cst</code>	The fence is both an acquire fence and a release fence, and is sequentially consistent.

atomic_store

Atomically stores a value in an atomic object.

```
template <class Ty>
inline Ty atomic_store_explicit(const volatile atomic<Ty>* Atom, Ty Value) noexcept;
template <class Ty>
inline Ty atomic_store_explicit(const atomic<Ty>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an atomic object that contains a value of type `Ty`.

Value

A value of type `Ty`.

Remarks

`atomic_store` stores *Value* in the object that is pointed to by *Atom*, within the `memory_order_seq_cst` [memory_order](#) constraint.

atomic_store_explicit

Atomically stores a value in an atomic object.

```
template <class Ty>
inline Ty atomic_store_explicit(
    const volatile atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;

template <class Ty>
inline Ty atomic_store_explicit(
    const atomic<Ty>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Value

A value of type `Ty`.

Order

A [memory_order](#). Do not use `memory_order_consume`, `memory_order_acquire`, or `memory_order_acq_rel`.

Remarks

`atomic_store` stores *Value* in the object that is pointed to by *Atom*, within the `memory_order` that is specified by *Order*.

atomic_thread_fence

Acts as a *fence*—which is a memory synchronization primitive that enforces ordering between load/store operations—without an associated atomic operation.

```
inline void atomic_thread_fence(memory_order Order) noexcept;
```

Parameters

Order

A memory ordering constraint that determines fence type.

Remarks

The *Order* argument determines fence type.

<code>memory_order_relaxed</code>	The fence has no effect.
<code>memory_order_consume</code>	The fence is an acquire fence.
<code>memory_order_acquire</code>	The fence is an acquire fence.
<code>memory_order_release</code>	The fence is a release fence.
<code>memory_order_acq_rel</code>	The fence is both an acquire fence and a release fence.

<div data-bbox="156 143 402 179" data-label="Text"> <pre>memory_order_seq_cst</pre> </div>	<div data-bbox="810 143 1420 208" data-label="Text"> <p>The fence is both an acquire fence and a release fence, and is sequentially consistent.</p> </div>
--	--

kill_dependency

Removes a dependency.

```
template <class Ty>
Ty kill_dependency(Ty Arg) noexcept;
```

Parameters

Arg

A value of type `Ty`.

Return Value

The return value is *Arg*. The evaluation of *Arg* does not carry a dependency to the function call. By breaking a possible dependency chain, the function might permit the compiler to generate more efficient code.

See also

[<atomic>](#)

<atomic> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

memory_order Enum

Supplies symbolic names for synchronization operations on memory locations. These operations affect how assignments in one thread become visible in another.

```
typedef enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst,  
} memory_order;
```

Enumeration members

<code>memory_order_relaxed</code>	No ordering required.
<code>memory_order_consume</code>	A load operation acts as a consume operation on the memory location.
<code>memory_order_acquire</code>	A load operation acts as an acquire operation on the memory location.
<code>memory_order_release</code>	A store operation acts as a release operation on the memory location.
<code>memory_order_acq_rel</code>	Combines <code>memory_order_acquire</code> and <code>memory_order_release</code> .
<code>memory_order_seq_cst</code>	Combines <code>memory_order_acquire</code> and <code>memory_order_release</code> . Memory accesses that are marked as <code>memory_order_seq_cst</code> must be sequentially consistent.

See also

[<atomic>](#)

<bitset>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template class `bitset` and two supporting template functions for representing and manipulating fixed-size sequences of bits.

Syntax

```
#include <bitset>
```

Operators

OPERATOR	DESCRIPTION
operator&	Performs a bitwise AND between two bitsets.
operator<<	Inserts a text representation of the bit sequence into the standard output stream.
operator>>	Inserts a text representation of the bit sequence into the standard input stream.
operator^	Performs a bitwise EXCLUSIVE-OR between two bitsets.
operator 	Performs a bitwise OR between two bitsets.

Classes

CLASS	DESCRIPTION
bitset Class	The template class describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<bitset> operators

11/8/2018 • 5 minutes to read • [Edit Online](#)

<code>operator&</code>	<code>operator>></code>	<code>operator<<</code>
<code>operator^</code>	<code>operator </code>	

operator&

Performs a bitwise `AND` between two bitsets.

```
template <size_t size>
bitset<size>
operator&(
    const bitset<size>& left,
    const bitset<size>& right);
```

Parameters

left

The first of the two bitsets whose respective elements are to be combined with the bitwise `AND`.

right

The second of the two valarrays whose respective elements are to be combined with the bitwise `AND`.

Return Value

A bitset whose elements are the result of performing the `AND` operation on the corresponding elements of *left* and *right*.

Example

```
// bitset_and.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    bitset<4> b1 ( string("0101") );
    bitset<4> b2 ( string("0011") );
    bitset<4> b3 = b1 & b2;
    cout << "bitset 1: " << b1 << endl;
    cout << "bitset 2: " << b2 << endl;
    cout << "bitset 3: " << b3 << endl;
}
```

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0001
```

operator<<

Inserts a text representation of the bit sequence into the output stream.

```
template <class CharType, class Traits, size_t N>
basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& ostr,
    const bitset<N>& right);
```

Parameters

right

An object of type **bitset<N>** that is to be inserted into the output stream as a string.

Return Value

A text representation of the bit sequence in `ostr`.

Remarks

The template function overloads `operator<<`, allowing a bitset to be written out without first converting it into a string. The template function effectively executes:

```
ostr << _Right.to_string < CharType, Traits, allocator< CharType> > > ( )
```

Example

```
// bitset_op_insert.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 9 );

    // bitset inserted into output stream directly
    cout << "The ordered set of bits in the bitset<5> b1(9)"
        << "\n can be output with the overloaded << as: ( "
        << b1 << " )" << endl;

    // Compare converting bitset to a string before
    // inserting it into the output stream
    string s1;
    s1 = b1.template to_string<char,
        char_traits<char>, allocator<char> >( );
    cout << "The string returned from the bitset b1"
        << "\n by the member function to_string( ) is: "
        << s1 << "." << endl;
}
```

operator>>

Reads a string of bit characters into a bitset.

```
template <class CharType, class Traits, size_t Bits>
basic_istream<CharType, Traits>& operator>> (
    basic_istream<CharType, Traits>&
    _Istr,
    bitset<N>&
    right);
```

Parameters

_Istr

The string that is entered into the input stream to be inserted into the bitset.

right

The bitset that is receiving the bits from the input stream.

Return Value

The template function returns the string *_Istr*.

Remarks

The template function overloads `operator>>` to store in the bitset *_Right* the value `bitset(str)`, where `str` is an object of type `basic_string < CharType, Traits, allocator< CharType> > &` extracted from *_Istr*.

The template function extracts elements from *_Istr* and inserts them into the bitset until:

- All the bit elements have been extracted from the input stream and stored in the bitset.
- The bitset is filled up with bits from the input stream.
- An input element is encountered that is neither 0 nor 1.

Example

```

#include <bitset>
#include <iostream>
#include <string>

using namespace std;
int main()
{

    bitset<5> b1;
    cout << "Enter string of (0 or 1) bits for input into bitset<5>.\n"
         << "Try bit string of length less than or equal to 5,\n"
         << " (for example: 10110): ";
    cin >> b1;

    cout << "The ordered set of bits entered from the "
         << "keyboard\n has been input into bitset<5> b1 as: ( "
         << b1 << " )" << endl;

    // Truncation due to longer string of bits than length of bitset
    bitset<2> b3;
    cout << "Enter string of bits (0 or 1) for input into bitset<2>.\n"
         << " Try bit string of length greater than 2,\n"
         << " (for example: 1011): ";
    cin >> b3;

    cout << "The ordered set of bits entered from the "
         << "keyboard\n has been input into bitset<2> b3 as: ( "
         << b3 << " )" << endl;

    // Flushing the input stream
    char buf[100];
    cin.getline(&buf[0], 99);

    // Truncation with non-bit value
    bitset<5> b2;
    cout << "Enter a string for input into bitset<5>.\n"
         << " that contains a character than is NOT a 0 or a 1,\n "
         << " (for example: 10k01): ";
    cin >> b2;

    cout << "The string entered from the keyboard\n"
         << " has been input into bitset<5> b2 as: ( "
         << b2 << " )" << endl;
}

```

operator[^]

Performs a bitwise **EXCLUSIVE-OR** between two bitsets.

```

template <size_t size>
bitset<size>
operator^(
    const bitset<size>& left,
    const bitset<size>& right);

```

Parameters

left

The first of the two bitsets whose respective elements are to be combined with the bitwise **EXCLUSIVE-OR** .

right

The second of the two valarrays whose respective elements are to be combined with the bitwise **EXCLUSIVE-OR** .

Return Value

A bitset whose elements are the result of performing the `EXCLUSIVE-OR` operation on the corresponding elements of *left* and *right*.

Example

```
// bitset_xor.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    bitset<4> b1 ( string("0101") );
    bitset<4> b2 ( string("0011") );
    bitset<4> b3 = b1 ^ b2;
    cout << "bitset 1: " << b1 << endl;
    cout << "bitset 2: " << b2 << endl;
    cout << "bitset 3: " << b3 << endl;
}
```

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0110
```

operator|

Performs a bitwise `OR` between two bitsets.

```
template <size_t size>
bitset<size>
operator|(
    const bitset<size>& left,
    const bitset<size>& right);
```

Parameters

left

The first of the two bitsets whose respective elements are to be combined with the bitwise `OR`.

right

The second of the two valarrays whose respective elements are to be combined with the bitwise `OR`.

Return Value

A bitset whose elements are the result of performing the `OR` operation on the corresponding elements of *left* and *right*.

Example

```
// bitset_or.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    bitset<4> b1 ( string("0101") );
    bitset<4> b2 ( string("0011") );
    bitset<4> b3 = b1 | b2;
    cout << "bitset 1: " << b1 << endl;
    cout << "bitset 2: " << b2 << endl;
    cout << "bitset 3: " << b3 << endl;
}
```

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0111
```

See also

[<bitset>](#)

bitset Class

3/28/2019 • 34 minutes to read • [Edit Online](#)

Describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions. The `bitset` class supports operations on objects of type `bitset` that contain a collection of bits and provide constant-time access to each bit.

Syntax

```
template <size_t N>
class bitset
```

Parameters

N

Specifies the number of bits in the `bitset` object with a nonzero integer of type `size_t` that must be known at compile time.

Remarks

Unlike the similar [vector<bool> Class](#), the `bitset` class does not have iterators and is not a C++ Standard Library container. It also differs from `vector<bool>` by being of some specific size that is fixed at compile time in accordance with the size specified by the template parameter *N* when the **`bitset<N>`** is declared.

A bit is set if its value is 1 and reset if its value is 0. To flip or invert a bit is to change its value from 1 to 0 or from 0 to 1. The *N* bits in a `bitset` are indexed by integer values from 0 to *N* - 1, where 0 indexes the first bit position and *N* - 1 the final bit position.

Constructors

CONSTRUCTOR	DESCRIPTION
bitset	Constructs an object of class <code>bitset<N></code> and initializes the bits to zero, to some specified value, or to values obtained from characters in a string.

Typedefs

TYPE NAME	DESCRIPTION
element_type	A type that is a synonym for the data type <code>bool</code> and can be used to reference element bits in a <code>bitset</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
all	Tests all of the bits in this <code>bitset</code> to determine whether they are all set to <code>true</code> .

MEMBER FUNCTION	DESCRIPTION
any	The member function tests whether any bit in the sequence is set to 1.
count	The member function returns the number of bits set in the bit sequence.
flip	Inverts the value of all the bits in a <code>bitset</code> or inverts a single bit at a specified position.
none	Tests if no bit has been set to 1 in a <code>bitset</code> object.
reset	Resets all the bits in a <code>bitset</code> to 0 or resets a bit at a specified position to 0.
set	Sets all the bits in a <code>bitset</code> to 1 or sets a bit at a specified position to 1.
size	Returns the number of bits in a <code>bitset</code> object.
test	Tests whether the bit at a specified position in a <code>bitset</code> is set to 1.
to_string	Converts a <code>bitset</code> object to a string representation.
to_ullong	Returns the sum of the bit values in the <code>bitset</code> as an unsigned long long .
to_ulong	Converts a <code>bitset</code> object to the unsigned long that would generate the sequence of bits contained if used to initialize the <code>bitset</code> .

Member Classes

MEMBER CLASS	DESCRIPTION
reference	A proxy class that provides references to bits contained in a <code>bitset</code> that is used to access and manipulate the individual bits as a helper class for the <code>operator[]</code> of class <code>bitset</code> .

Operators

OPERATOR	DESCRIPTION
operator!=	Tests a target <code>bitset</code> for inequality with a specified <code>bitset</code> .
operator&=	Performs a bitwise combination of bitsets with the logical AND operation.
operator<<	Shifts the bits in a <code>bitset</code> to the left a specified number of positions and returns the result to a new <code>bitset</code> .

OPERATOR	DESCRIPTION
<code>operator<<=</code>	Shifts the bits in a <code>bitset</code> to the left a specified number of positions and returns the result to the targeted <code>bitset</code> .
<code>operator==</code>	Tests a target <code>bitset</code> for equality with a specified <code>bitset</code> .
<code>operator>></code>	Shifts the bits in a <code>bitset</code> to the right a specified number of positions and returns the result to a new <code>bitset</code> .
<code>operator>>=</code>	Shifts the bits in a <code>bitset</code> to the right a specified number of positions and returns the result to the targeted <code>bitset</code> .
<code>operator[]</code>	Returns a reference to a bit at a specified position in a <code>bitset</code> if the <code>bitset</code> is modifiable; otherwise, it returns the value of the bit at that position.
<code>operator^=</code>	Performs a bitwise combination of bitsets with the exclusive <code>OR</code> operation.
<code>operator =</code>	Performs a bitwise combination of bitsets with the inclusive <code>OR</code> operation.
<code>operator~</code>	Inverts all the bits in a target <code>bitset</code> and returns the result.

Requirements

Header: `<bitset>`

Namespace: `std`

`bitset::all`

Tests all of the bits in this `bitset` to determine if they are all set to true.

```
bool all() const;
```

Return Value

Returns true if all bits in this set are true. Returns **false** if one or more bits are false.

`bitset::any`

Tests whether any bit in the sequence is set to 1.

```
bool any() const;
```

Return Value

true if any bit in the `bitset` is set to 1; **false** if all the bits are 0.

Example

```

// bitset_any.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 6 );
    bool b, rb;

    cout << "The original bitset b1( 6 ) is: ( "<< b1 << " )"
         << endl;

    b = b1.any ( );

    if ( b )
        cout << "At least one of the bits in bitset is set to 1."
             << endl;
    else
        cout << "None of the bits in bitset are set to 1."
             << endl;

    bitset<5> rb1;
    rb1 = b1.reset ( );

    cout << "The reset bitset is: ( "<< b1 << " )"
         << endl;

    rb = rb1.any ( );

    if ( rb )
        cout << "At least one of the bits in the reset bitset "
             << "are set to 1." << endl;
    else
        cout << "None of the bits in bitset b1 are set to 1."
             << endl;
}

```

```

The original bitset b1( 6 ) is: ( 00110 )
At least one of the bits in bitset is set to 1.
The reset bitset is: ( 00000 )
None of the bits in bitset b1 are set to 1.

```

bitset::bitset

Constructs an object of class `bitset<N>` and initializes the bits to zero, or to some specified value, or to values obtained from characters in a string.

```

bitset();

bitset(
    unsigned long long val);

explicit bitset(
    const char* _CStr);

template <class CharType,
    class Traits,
    class Allocator>
explicit bitset(
    const basic_string<CharType, Traits, Allocator>& str,
    typename basic_string<CharType, Traits, Allocator>::size_type _Pos = 0);

template <class CharType,
    class Traits,
    class Allocator>
explicit bitset(
    const basic_string<CharType, Traits, Allocator>& str,
    typename basic_string<CharType, Traits, Allocator>::size_type _Pos,
    typename basic_string<CharType, Traits, Allocator>::size_type count,
    CharType _Zero = CharType ('0'),
    CharType _One = CharType ('1'));

```

Parameters

val

The unsigned integer whose base two representation is used to initialize the bits in the bitset being constructed.

str

The string of zeros and ones used to initialize the bitset bit values.

_CStr

A C-style string of zeros and ones used to initialize the bitset bit values.

_Pos

The position of the character in the string, counting from left to right and starting with zero, used to initialize the first bit in the bitset.

count

The number of characters in the string that is used to provide initial values for the bits in the bitset.

_Zero

The character that is used to represent a zero. The default is '0'.

_One

The character that is used to represent a one. The default is '1'.

Remarks

Three constructors can be used to construct objects of class `bitset<N>`:

- The first constructor accepts no parameters, constructs an object of class `bitset<N>` and initializes all N bits to a default value of zero.
- The second constructor constructs an object of class `bitset<N>` and initializes the bits by using the single **unsigned long long** parameter.
- The third constructor constructs an object of class `bitset<N>`, initializing the N bits to values that correspond to the characters provided in a c-style character string of zeros and ones. You call the constructor without casting the string into a string type: `bitset<5> b5("01011");`

There are also two constructor templates provided:

- The first constructor template constructs an object of class `bitset<N>` and initializes bits from the characters provided in a string of zeros and ones. If any characters of the string are other than 0 or 1, the constructor throws an object of class `invalid_argument`. If the position specified (`_Pos`) is beyond the length of the string, then the constructor throws an object of class `out_of_range`. The constructor sets only those bits at position j in the bitset for which the character in the string at position `_Pos + j` is 1. By default, `_Pos` is 0.
- The second constructor template is similar to the first, but includes an additional parameter (`count`) that is used to specify the number of bits to initialize. It also has two optional parameters, `_Zero` and `_One`, which indicate what character in `str` is to be interpreted to mean a 0 bit and a 1 bit, respectively.

Example


```

// bitset_bitset.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    // Using the default constructor
    using namespace std;
    bitset<2> b0;
    cout << "The set of bits in bitset<2> b0 is: ( "
         << b0 << " )." << endl;

    // Using the second member function
    bitset<5> b1 ( 6 );
    cout << "The set of bits in bitset<5> b1( 6 ) is: ( "
         << b1 << " )." << endl;

    // The template parameter N can be an expression
    bitset< 2 * sizeof ( int ) > b2;
    cout << "The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( "
         << b2 << " )." << endl;

    // The base two representation will be truncated
    // if its length exceeds the size of the bitset
    bitset<3> b3 ( 6 );
    cout << "The set of bits in bitset<3> b3( 6 ) is ( "
         << b3 << " )." << endl;

    // Using a c-style string to initialize the bitset
    bitset<7> b3andahalf ( "1001001" );
    cout << "The set of bits in bitset<7> b3andahalf ( \"1001001\" )"
         << " is ( " << b3andahalf << " )." << endl;

    // Using the fifth member function with the first parameter
    string bitval4 ( "10011" );
    bitset<5> b4 ( bitval4 );
    cout << "The set of bits in bitset<5> b4( bitval4 ) is ( "
         << b4 << " )." << endl;

    // Only part of the string may be used for initialization

    // Starting at position 3 for a length of 6 (100110)
    string bitval5 ("11110011011");
    bitset<6> b5 ( bitval5, 3, 6 );
    cout << "The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( "
         << b5 << " )." << endl;

    // The bits not initialized with part of the string
    // will default to zero
    bitset<11> b6 ( bitval5, 3, 5 );
    cout << "The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( "
         << b6 << " )." << endl;

    // Starting at position 2 and continue to the end of the string
    bitset<9> b7 ( bitval5, 2 );
    cout << "The set of bits in bitset<9> b7( bitval, 2 ) is ( "
         << b7 << " )." << endl;
}

```

```
The set of bits in bitset<2> b0 is: ( 00 ).
The set of bits in bitset<5> b1( 6 ) is: ( 00110 ).
The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( 00000000 ).
The set of bits in bitset<3> b3( 6 ) is ( 110 ).
The set of bits in bitset<5> b4( bitval4 ) is ( 10011 ).
The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( 100110 ).
The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( 00000010011 ).
The set of bits in bitset<9> b7( bitval, 2 ) is ( 110011011 ).
```

bitset::count

Returns the number of bits set in the bit sequence.

```
size_t count() const;
```

Return Value

The number of bits set in the bit sequence.

Example

The following example demonstrates the use of the `bitset::count` member function.

```
// bitset_count.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1(4);

    cout << "The collection of bits in the original bitset is: ( "
         << b1 << " )" << endl;

    size_t i;
    i = b1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << i << "." << endl;

    bitset<5> fb1;
    fb1 = b1.flip();

    cout << "The collection of flipped bits in the modified bitset "
         << "is: ( " << fb1 << " )" << endl;

    size_t ii;
    ii = fb1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << ii << "." << endl;
}
```

```
The collection of bits in the original bitset is: ( 00100 )
The number of bits in the bitset set to 1 is: 1.
The collection of flipped bits in the modified bitset is: ( 11011 )
The number of bits in the bitset set to 1 is: 4.
```

bitset::element_type

A type that is a synonym for the data type **bool** and can be used to reference element bits in a bitset.

```
typedef bool element_type;
```

Example

```
// bitset_elem_type.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<3> b1 ( 2 );
    cout << "Original bitset b1(6) is: ( "<< b1 << " )"
         << endl;

    //Compare two ways to reference bits in a bitset
    bool b;
    bitset<5>::element_type e;

    b = b1.test ( 2 );
    if ( b )
        cout << "The bit at position 2 of bitset b1"
             << "has a value of 1." << endl;
    else
        cout << "The bit at position 2 of bitset b1"
             << "has a value of 0." << endl;
    b1[2] = 1;
    cout << "Bitset b1 modified by b1[2] = 1 is: ( "<< b1 << " )"
         << endl;

    e = b1.test ( 2 );
    if ( e )
        cout << "The bit at position 2 of bitset b1"
             << "has a value of 1." << endl;
    else
        cout << "The bit at position 2 of bitset b1"
             << "has a value of 0." << endl;
}
```

```
Original bitset b1(6) is: ( 010 )
The bit at position 2 of bitset b1has a value of 0.
Bitset b1 modified by b1[2] = 1 is: ( 110 )
The bit at position 2 of bitset b1has a value of 1.
```

bitset::flip

Inverts the value of all the bits in a bitset or inverts a single bit at a specified position.

```
bitset<N>& flip();
bitset<N>& flip(size_t _Pos);
```

Parameters

_Pos

The position of the bit whose value is to be inverted.

Return Value

A copy of the modified bitset for which the member function was invoked.

Remarks

The second member function throws an [out_of_range](#) exception if the position specified as a parameter is greater than the size N of the **bitset**< N > whose bit was inverted.

Example

```
// bitset_flip.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 6 );

    cout << "The collection of bits in the original bitset is: ( "
         << b1 << " )" << endl;

    bitset<5> fb1;
    fb1 = b1.flip ( );

    cout << "After flipping all the bits, the bitset becomes: ( "
         << fb1 << " )" << endl;

    bitset<5> f3b1;
    f3b1 = b1.flip ( 3 );

    cout << "After flipping the fourth bit, the bitset becomes: ( "
         << f3b1 << " )" << endl << endl;

    bitset<5> b2;
    int i;
    for ( i = 0 ; i <= 4 ; i++ )
    {
        b2.flip(i);
        cout << b2 << " The bit flipped is in position "
             << i << ".\n";
    }
}
```

```
The collection of bits in the original bitset is: ( 00110 )
After flipping all the bits, the bitset becomes: ( 11001 )
After flipping the fourth bit, the bitset becomes: ( 10001 )
```

```
00001 The bit flipped is in position 0.
00011 The bit flipped is in position 1.
00111 The bit flipped is in position 2.
01111 The bit flipped is in position 3.
11111 The bit flipped is in position 4.
```

bitset::none

Tests if no bit has been set to 1 in a bitset object.

```
bool none() const;
```

Return Value

true if no bit in the bitset has been set to 1; **false** if at least one bit has been set to 1.

Example

```
// bitset_none.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 6 );
    bool b, rb;

    cout << "Original bitset b1(6) is: ( " << b1 << " )"
         << endl;

    b = b1.none ( );

    if ( b )
        cout << "None of the bits in bitset b1 are set to 1."
              << endl;
    else
        cout << "At least one of the bits in bitset b1 is set to 1."
              << endl;

    bitset<5> rb1;
    rb1 = b1.reset ( );
    rb = rb1.none ( );
    if ( rb )
        cout << "None of the bits in bitset b1 are set to 1."
              << endl;
    else
        cout << "At least one of the bits in bitset b1 is set to 1."
              << endl;
}
```

```
Original bitset b1(6) is: ( 00110 )
At least one of the bits in bitset b1 is set to 1.
None of the bits in bitset b1 are set to 1.
```

bitset::operator!=

Tests a target bitset for inequality with a specified bitset.

```
bool operator!=(const bitset<N>& right) const;
```

Parameters

right

The bitset that is to be compared to the target bitset for inequality.

Return Value

true if the bitsets are different; **false** if they are the same.

Remarks

Bitsets must be of the same size to be tested for inequality by the member operator function.

Example

```
// bitset_op_NE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );
    bitset<5> b2 ( 7 );
    bitset<5> b3 ( 2 );
    bitset<4> b4 ( 7 );

    if ( b1 != b2 )
        cout << "Bitset b1 is different from bitset b2." << endl;
    else
        cout << "Bitset b1 is the same as bitset b2." << endl;

    if ( b1 != b3 )
        cout << "Bitset b1 is different from bitset b3." << endl;
    else
        cout << "Bitset b1 is the same as bitset b3." << endl;

    // This would cause an error because bitsets must have the
    // same size to be tested
    // if ( b1 != b4 )
    //     cout << "Bitset b1 is different from bitset b4." << endl;
    // else
    //     cout << "Bitset b1 is the same as bitset b4." << endl;
}
```

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

bitset::operator&=

Performs a bitwise combination of bitsets with the logical **AND** operation.

```
bitset<N>& operator&=(const bitset<N>& right);
```

Parameters

right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise **AND** operation with the bitset specified as a parameter.

Remarks

Two bits combined by the **AND** operator return **true** if each bit is true; otherwise, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the **AND** operator by the member operator function.

Example

```

// bitset_op_bitwise.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );
    bitset<5> b2 ( 11 );
    bitset<4> b3 ( 7 );

    cout << "The target bitset b1 is:   ( "<< b1 << " )." << endl;
    cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
    cout << endl;

    b1 &= b2;
    cout << "After bitwise AND combination,\n"
        << "the target bitset b1 becomes:  ( "<< b1 << " )."
        << endl;

    // Note that the parameter-specified bitset is unchanged
    cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

    // The following would cause an error because the bisets
    // must be of the same size to be combined
    // b1 &= b3;
}

```

```

The target bitset b1 is:   ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise AND combination,
the target bitset b1 becomes:  ( 00011 ).
The parameter bitset b2 remains: ( 01011 ).

```

bitset::operator<<

Shifts the bits in a bitset to the left a specified number of positions and returns the result to a new bitset.

```

bitset<N> operator<<(size_t _Pos) const;

```

Parameters

_Pos

The number of positions to the left that the bits in the bitset are to be shifted.

Return Value

The modified bitset with the bits shifted to the left the required number of positions.

Remarks

The member operator function returns **bitset(*this) <<= pos**, where **<<=** shifts the bits in a bitset to the left a specified number of positions and returns the result to the targeted bitset.

Example

```

// bitset_op_LS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );

    cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

    bitset<5> b2;
    b2 = b1 << 2;

    cout << "After shifting the bits 2 positions to the left,\n"
        << " the bitset b2 is: ( "<< b2 << " )."
        << endl;

    bitset<5> b3 = b2 >> 1;

    cout << "After shifting the bits 1 position to the right,\n"
        << " the bitset b3 is: ( " << b3 << " )."
        << endl;
}

```

bitset::operator<<=

Shifts the bits in a bitset to the left a specified number of positions and returns the result to the targeted bitset.

```

bitset<N>& operator<<=(size_t _Pos);

```

Parameters

_Pos

The number of positions to the left the bits in the bitset are to be shifted.

Return Value

The targeted bitset modified so that the bits have been shifted to the left the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

```

// bitset_op_LSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 7 );
    cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;
    b1 <<= 2;
    cout << "After shifting the bits 2 positions to the left,\n"
        << "the target bitset b1 becomes: ( "<< b1 << " )."
        << endl;
}

```


The target bitset b1 is: (00111).
After shifting the bits 2 positions to the left,
the target bitset b1 becomes: (11100).

bitset::operator==

Tests a target bitset for equality with a specified bitset.

```
bool operator==(const bitset<N>& right) const;
```

Parameters

right

The bitset that is to be compared to the target bitset for equality.

Return Value

true if the bitsets are the same; **false** if they are different.

Remarks

Bitsets must be of the same size to be tested for equality by the member operator function.

Example

```
// bitset_op_EQ.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 7 );
    bitset<5> b2 ( 7 );
    bitset<5> b3 ( 2 );
    bitset<4> b4 ( 7 );

    if ( b1 == b2 )
        cout << "Bitset b1 is the same as bitset b2." << endl;
    else
        cout << "Bitset b1 is different from bitset b2." << endl;

    if ( b1 == b3 )
        cout << "Bitset b1 is the same as bitset b3." << endl;
    else
        cout << "Bitset b1 is different from bitset b3." << endl;

    // This would cause an error because bitsets must have the
    // same size to be tested
    // if ( b1 == b4 )
    //     cout << "Bitset b1 is the same as bitset b4." << endl;
    // else
    //     cout << "Bitset b1 is different from bitset b4." << endl;
}
```

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

bitset::operator>>

Shifts the bits in a bitset to the right a specified number of positions and returns the result to a new bitset.

```
bitset<N> operator>>(size_t _Pos) const;
```

Parameters

_Pos

The number of positions to the right the bits in the bitset are to be shifted.

Return Value

A new bitset where the bits have been shifted to the right the required number of positions relative to the targeted bitset.

Example

```
// bitset_op_RS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 7 );
    cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

    bitset<5> b2;
    b2 = b1 << 2;

    cout << "After shifting the bits 2 positions to the left,\n"
         << "the bitset b2 is: ( "<< b2 << " )."
         << endl;
    bitset<5> b3 = b2 >> 1;

    cout << "After shifting the bits 1 position to the right,\n"
         << "the bitset b3 is: ( " << b3 << " )."
         << endl;
}
```

```
The bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
the bitset b2 is: ( 11100 ).
After shifting the bits 1 position to the right,
the bitset b3 is: ( 01110 ).
```

bitset::operator>>=

Shifts the bits in a bitset to the right a specified number of positions and returns the result to the targeted bitset.

```
bitset<N>& operator>>=(size_t _Pos);
```

Parameters

_Pos

The number of positions to the right the bits in the bitset are to be shifted.

Return Value

The targeted bitset modified so that the bits have been shifted to the right the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

```
// bitset_op_RSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 28 );
    cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;

    b1 >>= 2;
    cout << "After shifting the bits 2 positions to the right,\n"
        << "the target bitset b1 becomes: ( "<< b1 << " )."
        << endl;
}
```

```
The target bitset b1 is: ( 11100 ).
After shifting the bits 2 positions to the right,
the target bitset b1 becomes: ( 00111 ).
```

bitset::operator[]

Returns a reference to a bit at a specified position in a bitset if the bitset is modifiable; otherwise, it returns the value of the bit at that position.

```
bool operator[](size_t _Pos) const;
reference operator[](size_t _Pos);
```

Parameters

_Pos

The position locating the bit within the bitset.

Remarks

When you define `_ITERATOR_DEBUG_LEVEL` as 1 or 2 in your build, a runtime error will occur in your executable if you attempt to access an element outside the bounds of the bitset. For more informations, see [Checked Iterators](#).

Example

```
// bitset_op_REF.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bool b;
    bitset<5> b1 ( 6 );
    cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
         << endl;

    int i;
    for ( i = 0 ; i <= 4 ; i++ )
    {
        b = b1[ i ];
        cout << " The bit in position "
             << i << " is " << b << ".\n";
    }
}
```

bitset::operator^=

Performs a bitwise combination of bitsets with the exclusive **OR** operation.

```
bitset<N>& operator^=(const bitset<N>& right);
```

Parameters

right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise exclusive **OR** operation with the bitset specified as a parameter.

Remarks

Two bits combined by the exclusive **OR** operator return **true** if at least one, but not both, of the bits is **true**; otherwise, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the exclusive **OR** operator by the member operator function.

Example

```

// bitset_op_bitwiseOR.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;
    bitset<5> b1 ( 7 );
    bitset<5> b2 ( 11 );
    bitset<4> b3 ( 7 );

    cout << "The target bitset b1 is:   ( "<< b1 << " )." << endl;
    cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
    cout << endl;

    b1 ^= b2;
    cout << "After bitwise exclusive OR combination,\n"
        << "the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

    // Note that the parameter-specified bitset is unchanged
    cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

    // The following would cause an error because the bitsets
    // must be of the same size to be combined
    // b1 |= b3;
}

```

```

The target bitset b1 is:   ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise exclusive OR combination,
the target bitset b1 becomes:   ( 01100 ).
The parameter bitset b2 remains: ( 01011 ).

```

bitset::operator|=

Performs a bitwise combination of bitsets with the inclusive `OR` operation.

```

bitset<N>& operator|=(const bitset<N>& right);

```

Parameters

right

The bitset that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise inclusive `OR` operation with the bitset specified as a parameter.

Remarks

Two bits combined by the inclusive `OR` operator return **true** if at least one of the bits is **true**; if both bits are **false**, their combination returns **false**.

Bitsets must be of the same size to be combined bitwise with the inclusive `OR` operator by the member operator function.

Example

```
// bitset_op_BIO.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );
    bitset<5> b2 ( 11 );
    bitset<4> b3 ( 7 );

    cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
    cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
    cout << endl;

    b1 |= b2;
    cout << "After bitwise inclusive OR combination,\n"
        << "the target bitset b1 becomes:  ( "<< b1 << " )."
        << endl;

    // Note that the parameter-specified bitset is unchanged
    cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

    // The following would cause an error because the bisets
    // must be of the same size to be combined
    // b1 |= b3;
}
```

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise inclusive OR combination,
the target bitset b1 becomes:  ( 01111 ).
The parameter bitset b2 remains: ( 01011 ).
```

bitset::operator~

Inverts all the bits in a target bitset and returns the result.

```
bitset<N> operator~() const;
```

Return Value

The bitset with all its bits inverted with respect to the targeted bitset.

Example

```

// bitset_op_invert.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <bitset>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );
    bitset<5> b2;
    b2 = ~b1;

    cout << "Bitset b1 is: ( "<< b1 << " )." << endl;
    cout << "Bitset b2 = ~b1 is: ( "<< b2 << " )." << endl;

    // These bits could also be flipped using the flip member function
    bitset<5> b3;
    b3 = b1.flip( );
    cout << "Bitset b3 = b1.flip( ) is: ( "<< b3 << " )." << endl;
}

```

```

Bitset b1 is: ( 00111 ).
Bitset b2 = ~b1 is: ( 11000 ).
Bitset b3 = b1.flip( ) is: ( 11000 ).

```

bitset::reference

A proxy class that provides references to bits contained in a `bitset` that is used to access and manipulate the individual bits as a helper class for the `operator[]` of class `bitset`.

```

class reference {
    friend class bitset<N>;
public:
    reference& operator=(bool val);
    reference& operator=(const reference& _Bitref);
    bool operator~() const;
    operator bool() const;
    reference& flip();
};

```

Parameters

val

The value of the object of type **bool** to be assigned to a bit in a `bitset`.

_Bitref

A reference of the form `x[i]` to the bit at position *i* in `bitset` *x*.

Return Value

A reference to the bit in the `bitset` specified by the argument position for the first, second, and fifth member functions of class `reference`, and **true** or **false**, to reflect the value of the modified bit in the `bitset` for the third and fourth member functions of class `reference`.

Remarks

The class `reference` exists only as a helper class for the `bitset` `operator[]`. The member class describes an object that can access an individual bit within a `bitset`. Let *b* be an object of type **bool**, *x* and *y* objects of type `bitset< N >`, and *i* and *j* valid positions within such an object. The notation `x[i]` references the bit at position *i* in `bitset` *x*. The

member functions of class `reference` provide, in order, the following operations:

OPERATION	DEFINITION
$x[i] = b$	Stores bool value b at bit position i in bitset x .
$x[i] = y[j]$	Stores the value of the bit $y[j]$ at bit position i in bitset x .
$b = \sim x[i]$	Stores the flipped value of the bit $x[i]$ in bool b .
$b = x[i]$	Stores the value of the bit $x[i]$ in bool b .
$x[i].\text{flip}()$	Stores the flipped value of the bit $x[i]$ back at bit position i in x .

Example


```

// bitset_reference.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 2 );
    bitset<5> b2 ( 6 );
    cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
        << endl;
    cout << "The initialized bitset<5> b2( 6 ) is: ( "<< b2 << " )."
        << endl;

    // Example of x [i] = b storing bool b at bit position i
    // in bitset x
    b1[ 0 ] = true;
    cout << "The bitset<5> b1 with the bit at position 0 set to 1"
        << "is: ( "<< b1 << " )" << endl;

    // Example of x [i] = y [j] storing the bool value of the
    // bit at position j in bitset y at bit position i in bitset x
    b2 [4] = b1 [0];      // b1 [0] = true
    cout << "The bitset<5> b2 with the bit at position 4 set to the "
        << "value\nof the bit at position 0 of the bit in "
        << "bitset<5> b1 is: ( "<< b2 << " )" << endl;

    // Example of b = ~x [i] flipping the value of the bit at
    // position i of bitset x and storing the value in an
    // object b of type bool
    bool b = ~b2 [4];      // b2 [4] = false
    if ( b )
        cout << "The value of the object b = ~b2 [4] "
            << "of type bool is true." << endl;
    else
        cout << "The value of the object b = ~b2 [4] "
            << "of type bool is false." << endl;

    // Example of b = x [i] storing the value of the bit at
    // position i of bitset x in the object b of type bool
    b = b2 [4];
    if ( b )
        cout << "The value of the object b = b2 [4] "
            << "of type bool is true." << endl;
    else
        cout << "The value of the object b = b2 [4] "
            << "of type bool is false." << endl;

    // Example of x [i] . flip ( ) toggling the value of the bit at
    // position i of bitset x
    cout << "Before flipping the value of the bit at position 4 in "
        << "bitset b2,\nit is ( "<< b2 << " )." << endl;
    b2 [4].flip( );
    cout << "After flipping the value of the bit at position 4 in "
        << "bitset b2,\nit becomes ( "<< b2 << " )." << endl;
    bool c;
    c = b2 [4].flip( );
    cout << "After a second flip, the value of the position 4 "
        << "bit in b2 is now: " << c << ".";
}

```

The initialized bitset<5> b1(2) is: (00010).
The initialized bitset<5> b2(6) is: (00110).
The bitset<5> b1 with the bit at position 0 set to 1 is: (00011)
The bitset<5> b2 with the bit at position 4 set to the value
of the bit at position 0 of the bit in bitset<5> b1 is: (10110)
The value of the object b = ~b2 [4] of type bool is false.
The value of the object b = b2 [4] of type bool is true.
Before flipping the value of the bit at position 4 in bitset b2,
it is (10110).
After flipping the value of the bit at position 4 in bitset b2,
it becomes (00110).
After a second flip, the value of the position 4 bit in b2 is now: 1.

bitset::reset

Resets all the bits in a bitset to 0 or resets a bit at a specified position to 0.

```
bitset<N>& reset();  
bitset<N>& reset(size_t _Pos);
```

Parameters

_Pos

The position of the bit in the bitset to be reset to 0.

Return Value

A copy of the bitset for which the member function was invoked.

Remarks

The second member function throws an [out_of_range](#) exception if the position specified is greater than the size of the bitset.

Example

```
// bitset_reset.cpp  
// compile with: /EHsc  
#include <bitset>  
#include <iostream>  
  
int main( )  
{  
    using namespace std;  
  
    bitset<5> b1 ( 13 );  
    cout << "The set of bits in bitset<5> b1(13) is: ( "<< b1 << " )"  
        << endl;  
  
    bitset<5> b1r3;  
    b1r3 = b1.reset( 2 );  
    cout << "The collection of bits obtained from resetting the\n"  
        << "third bit of bitset b1 is: ( "<< b1r3 << " )"  
        << endl;  
  
    bitset<5> b1r;  
    b1r = b1.reset( );  
    cout << "The collection of bits obtained from resetting all\n"  
        << "the elements of the bitset b1 is: ( "<< b1r << " )"  
        << endl;  
}
```

```
The set of bits in bitset<5> b1(13) is: ( 01101 )
The collection of bits obtained from resetting the
third bit of bitset b1 is: ( 01001 )
The collection of bits obtained from resetting all
the elements of the bitset b1 is: ( 00000 )
```

bitset::set

Sets all the bits in a bitset to 1 or sets a bit at a specified position to 1.

```
bitset<N>& set();

bitset<N>& set(
    size_t _Pos,
    bool val = true);
```

Parameters

_Pos

The position of the bit in the bitset to be set to assigned a value.

val

The value to be assigned to the bit at the position specified.

Return Value

A copy of the bitset for which the member function was invoked.

Remarks

The second member function throws an [out_of_range](#) exception if the position specified is greater than the size of the bitset.

Example

```
// bitset_set.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 6 );
    cout << "The set of bits in bitset<5> b1(6) is: ( "<< b1 << " )"
         << endl;

    bitset<5> b1s0;
    b1s0 = b1.set( 0 );
    cout << "The collection of bits obtained from setting the\n"
         << "zeroth bit of bitset b1 is: ( "<< b1s0 << " )"
         << endl;

    bitset<5> bs1;
    bs1 = b1.set( );
    cout << "The collection of bits obtained from setting all the\n"
         << "elements of the bitset b1 is: ( "<< bs1 << " )"
         << endl;
}
```

```
The set of bits in bitset<5> b1(6) is: ( 00110 )
The collection of bits obtained from setting the
zeroth bit of bitset b1 is: ( 00111 )
The collection of bits obtained from setting all the
elements of the bitset b1 is: ( 11111 )
```

bitset::size

Returns the number of bits in a bitset object.

```
size_t size() const;
```

Return Value

The number of bits, N , in a bitset< N >.

Example

The following example demonstrates the use of the bitset::size member function.

```
// bitset_size.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main()
{
    using namespace std;

    bitset<5> b1(6);
    size_t i;

    cout << "The set of bits in bitset<5> b1( 6 ) is: ( "<< b1 << " )"
         << endl;

    i = b1.size();

    cout << "The number of bits in bitset b1 is: " << i << "."
         << endl;
}
```

```
The set of bits in bitset<5> b1( 6 ) is: ( 00110 )
The number of bits in bitset b1 is: 5.
```

bitset::test

Tests whether the bit at a specified position in a bitset is set to 1.

```
bool test(size_t _Pos) const;
```

Parameters

_Pos

The position of the bit in the bitset to be tested for its value.

Return Value

true if the bit specified by the argument position is set to 1; otherwise, **false**.

Remarks

The member function throws an [out_of_range](#)

bitset::to_string

Converts a bitset object to a string representation.

```
template <class charT = char, class traits = char_traits<charT>, class Allocator = allocator<charT> >
    basic_string<charT, traits, Allocator> to_string(charT zero = charT{'0'}, charT one = charT{'1'}) const;
```

Return value

A string object of class `basic_string`, where each bit set in the bitset has a corresponding character of 1, and a character of 0 if the bit is unset.

Example

```
// bitset_to_string.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );

    cout << "The ordered set of bits in the bitset<5> b1( 7 )"
         << "\n  that was generated by the number 7 is: ( "
         << b1 << " )" << endl;

    string s1;
    s1 = b1.template to_string<char,
char_traits<char>, allocator<char> >( );
    cout << "The string returned from the bitset b1"
         << "\n  by the member function to_string( ) is: "
         << s1 << "." << endl;
}
```

```
The ordered set of bits in the bitset<5> b1( 7 )
  that was generated by the number 7 is: ( 00111 )
The string returned from the bitset b1
  by the member function to_string( ) is: 00111.
```

bitset::to_ullong

Returns an **unsigned long long** value that contains the same bits set as the contents of the bitset object.

```
unsigned long long to_ullong() const;
```

Return value

Returns the sum of the bit values that are in the bit sequence as an **unsigned long long**. This **unsigned long long** value would re-create the same set bits if it is used to initialize a bitset.

Exceptions

Throws an [overflow_error](#) object if any bit in the bit sequence has a bit value that cannot be represented as a value of type **unsigned long long**.

Remarks

Returns the sum of the bit values that are in the bit sequence as an **unsigned long long**.

bitset::to_ulong

Converts a bitset object to the integer that would generate the sequence of bits contained if used to initialize the bitset.

```
unsigned long to_ulong( ) const;
```

Return value

An integer that would generate the bits in a bitset if used in the initialization of the bitset.

Remarks

Applying the member function would return the integer that has the same sequence of 1 and 0 digits as is found in sequence of bits contained in the bitset.

The member function throws an [overflow_error](#) object if any bit in the bit sequence has a bit value that cannot be represented as a value of type **unsigned long**.

Example

```
// bitset_to_ulong.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1 ( 7 );

    cout << "The ordered set of bits in the bitset<5> b1( 7 )"
         << "\n  that was generated by the number 7 is: ( "
         << b1 << " )" << endl;

    unsigned long int i;
    i = b1.to_ulong( );
    cout << "The integer returned from the bitset b1,"
         << "\n  by the member function to_ulong( ), that"
         << "\n  generated the bits as a base two number is: "
         << i << "." << endl;
}
```

```
The ordered set of bits in the bitset<5> b1( 7 )
  that was generated by the number 7 is: ( 00111 )
The integer returned from the bitset b1,
  by the member function to_ulong( ), that
  generated the bits as a base two number is: 7.
```

See also

[<bitset>](#)

<cassert>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header `<assert.h>` and adds the associated names to the `std` namespace.

Syntax

```
#include <cassert>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[assert Macro, _assert, _wassert](#)

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<complex>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the C++ Standard Library header [<complex>](#), which effectively includes the Standard C library header [<complex.h>](#) and adds the associated names to the `std` namespace.

Syntax

```
#include <complex>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

The name `clog`, which is declared in [<complex.h>](#), is not defined in the `std` namespace because of potential conflicts with the `clog` that is declared in [<iostream>](#).

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

<cctype>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <ctype.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cctype>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cerrno>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <errno.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cerrno>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cfenv>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <fenv.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cfenv>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<float>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <float.h>.

Syntax

```
#include <float>
```

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<chrono>

5/7/2019 • 3 minutes to read • [Edit Online](#)

Include the standard header <chrono> to define classes and functions that represent and manipulate time durations and time instants.

Beginning in Visual Studio 2015, the implementation of `steady_clock` has changed to meet the C++ Standard requirements for steadiness and monotonicity. `steady_clock` is now based on `QueryPerformanceCounter()` and `high_resolution_clock` is now a typedef for `steady_clock`. As a result, in the Microsoft C++ compiler `steady_clock::time_point` is now a typedef for `chrono::time_point<steady_clock>`; however, this is not necessarily the case for other implementations.

Syntax

```
#include <chrono>
```

Classes

NAME	DESCRIPTION
duration Class	Describes a type that holds a time interval.
time_point Class	Describes a type that represents a point in time.

Structs

NAME	DESCRIPTION
common_type Structure	Describes specializations of template class common_type for instantiations of <code>duration</code> and <code>time_point</code> .
duration_values Structure	Provides specific values for the <code>duration</code> template parameter <code>Rep</code> .
steady_clock struct	Represents a <code>steady</code> clock.
system_clock Structure	Represents a <i>clock type</i> that is based on the real-time clock of the system.
treat_as_floating_point Structure	Specifies whether a type can be treated as a floating-point type.

Functions

NAME	DESCRIPTION
duration_cast	Casts a <code>duration</code> object to a specified type.
time_point_cast	Casts a <code>time_point</code> object to a specified type.

Operators

NAME	DESCRIPTION
<code>operator-</code>	Operator for subtraction or negation of <code>duration</code> and <code>time_point</code> objects.
<code>operator!=</code>	Inequality operator that is used with <code>duration</code> or <code>time_point</code> objects.
<code>operator modulo</code>	Operator for modulo operations on <code>duration</code> objects.
<code>operator*</code>	Multiplication operator for <code>duration</code> objects.
<code>operator/</code>	Division operator for <code>duration</code> objects.
<code>operator+</code>	Adds <code>duration</code> and <code>time_point</code> objects.
<code>operator<</code>	Determines whether one <code>duration</code> or <code>time_point</code> object is less than another <code>duration</code> or <code>time_point</code> object.
<code>operator<=</code>	Determines whether one <code>duration</code> or <code>time_point</code> object is less than or equal to another <code>duration</code> or <code>time_point</code> object.
<code>operator==</code>	Determines whether two <code>duration</code> objects represent time intervals that have the same length, or whether two <code>time_point</code> objects represent the same point in time.
<code>operator></code>	Determines whether one <code>duration</code> or <code>time_point</code> object is greater than another <code>duration</code> or <code>time_point</code> object.
<code>operator>=</code>	Determines whether one <code>duration</code> or <code>time_point</code> object is greater than or equal to another <code>duration</code> or <code>time_point</code> object.

Predefined Duration Types

For more information about ratio types that are used in the following typedefs, see [<ratio>](#).

TYPEDEF	DESCRIPTION
<code>typedef duration<long long, nano> nanoseconds;</code>	Synonym for a <code>duration</code> type that has a tick period of one nanosecond.
<code>typedef duration<long long, micro> microseconds;</code>	Synonym for a <code>duration</code> type that has a tick period of one microsecond.
<code>typedef duration<long long, milli> milliseconds;</code>	Synonym for a <code>duration</code> type that has a tick period of one millisecond.
<code>typedef duration<long long> seconds;</code>	Synonym for a <code>duration</code> type that has a tick period of one second.

TYPEDEF	DESCRIPTION
<code>typedef duration<int, ratio<60> > minutes;</code>	Synonym for a <code>duration</code> type that has a tick period of one minute.
<code>typedef duration<int, ratio<3600> > hours;</code>	Synonym for a <code>duration</code> type that has a tick period of one hour.

Literals

(C++11) The `<chrono>` header defines the following [user-defined literals](#) that you can use for greater convenience, type-safety and maintainability of your code. These literals are defined in the `literals::chrono_literals` inline namespace and are in scope when `std::chrono` is in scope.

LITERAL	DESCRIPTION
<code>chrono::hours operator "" h(unsigned long long Val)</code>	Specifies hours as an integral value.
<code>chrono::duration<double, ratio<3600> > operator "" h(long double Val)</code>	Specifies hours as a floating-point value.
<code>chrono::minutes (operator "" min)(unsigned long long Val)</code>	Specifies minutes as an integral value.
<code>chrono::duration<double, ratio<60> > (operator "" min)(long double Val)</code>	Specifies minutes as a floating-point value.
<code>chrono::seconds operator "" s(unsigned long long Val)</code>	Specifies minutes as an integral value.
<code>chrono::duration<double> operator "" s(long double Val)</code>	Specifies seconds as a floating-point value.
<code>chrono::milliseconds operator "" ms(unsigned long long Val)</code>	Specifies milliseconds as an integral value.
<code>chrono::duration<double, milli> operator "" ms(long double Val)</code>	Specifies milliseconds as a floating-point value.
<code>chrono::microseconds operator "" us(unsigned long long Val)</code>	Specifies microseconds as an integral value.
<code>chrono::duration<double, micro> operator "" us(long double Val)</code>	Specifies microseconds as a floating-point value.
<code>chrono::nanoseconds operator "" ns(unsigned long long Val)</code>	Specifies nanoseconds as an integral value.
<code>chrono::duration<double, nano> operator "" ns(long double Val)</code>	Specifies nanoseconds as a floating-point value.

The following examples show how to use the chrono literals.

```
constexpr auto day = 24h;
constexpr auto week = 24h* 7;
constexpr auto my_duration_unit = 108ms;
```

Remarks

See also

[Header Files Reference](#)

<chrono> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

[duration_cast](#)

[time_point_cast](#)

duration_cast

Casts a `duration` object to a specified type.

```
template <class To, class Rep, class Period>
constexpr To duration_cast(const duration<Rep, Period>& Dur);
```

Return Value

A `duration` object of type `To` that represents the time interval `Dur`, which is truncated if it has to fit into the target type.

Remarks

If `To` is an instantiation of `duration`, this function does not participate in overload resolution.

time_point_cast

Casts a `time_point` object to a specified type.

```
template <class To, class Clock, class Duration>
time_point<Clock, To> time_point_cast(const time_point<Clock, Duration>& Tp);
```

Return Value

A `time_point` object that has a duration of type `To`.

Remarks

Unless `To` is an instantiation of `duration`, this function does not participate in overload resolution.

See also

[<chrono>](#)

<chrono> operators

10/31/2018 • 6 minutes to read • [Edit Online](#)

operator modulo	operator!=	operator>
operator>=	operator<	operator<=
operator*	operator+	operator-
operator/	operator==	

operator-

Operator for subtraction or negation of [duration](#) and [time_point](#) objects.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
operator-(
    const duration<Rep1, Period1>& Left,
    cconst duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
operator-(
    const time_point<Clock, Duration1>& Time,
    const duration<Rep2, Period2>& Dur);

template <class Clock, class Duration1, class Duration2>
constexpr typename common_type<Duration1, Duration2>::type
operator-(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Time

A `time_point` object.

Dur

A `duration` object.

Return Value

The first function returns a `duration` object whose interval length is the difference between the time intervals of the two arguments.

The second function returns a `time_point` object that represents a point in time that is displaced, by the negation of the time interval that is represented by *Dur*, from the point in time that is specified by *Time*.

The third function returns a `duration` object that represents the time interval between *Left* and *Right*.

operator!=

Inequality operator for `duration` or `time_point` objects.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Duration2>
constexpr bool operator!=(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

Each function returns `!(Left == Right)`.

operator*

Multiplication operator for `duration` objects.

```
template <class Rep1, class Period1, class Rep2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period1>
operator*(
    const duration<Rep1, Period1>& Dur,
    const Rep2& Mult);

template <class Rep1, class Rep2, class Period2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period2>
operator*(
    const Rep1& Mult,
    const duration<Rep2,
    Period2>& Dur);
```

Parameters

Dur

A `duration` object.

Mult

An integral value.

Return Value

Each function returns a `duration` object whose interval length is *Mult* multiplied by the length of *Dur*.

Unless `is_convertible<Rep2, common_type<Rep1, Rep2>>` holds true, the first function does not participate in overload resolution. For more information, see [<type_traits>](#).

Unless `is_convertible<Rep1, common_type<Rep1, Rep2>>` holds true, the second function does not participate in

overload resolution. For more information, see [<type_traits>](#).

operator/

Division operator for [duration](#) objects.

```
template <class Rep1, class Period1, class Rep2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period1>
operator/(
    const duration<Rep1, Period1>& Dur,
    const Rep2& Div);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<Rep1, Rep2>::type
operator/(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);
```

Parameters

Dur

A `duration` object.

Div

An integral value.

Left

The left `duration` object.

Right

The right `duration` object.

Return Value

The first operator returns a duration object whose interval length is the length of *Dur* divided by the value *Div*.

The second operator returns the ratio of the interval lengths of *Left* and *Right*.

Unless `is_convertible<Rep2, common_type<Rep1, Rep2>>` *holds true*, and `Rep2` is not an instantiation of `duration`, the first operator does not participate in overload resolution. For more information, see [<type_traits>](#).

operator+

Adds [duration](#) and [time_point](#) objects.

```

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
operator+(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
operator+(
    const time_point<Clock, Duration1>& Time,
    const duration<Rep2, Period2>& Dur);

template <class Rep1, class Period1, class Clock, class Duration2>
time_point<Clock, constexpr typename common_type<duration<Rep1, Period1>, Duration2>::type>
operator+(
    const duration<Rep1, Period1>& Dur,
    const time_point<Clock, Duration2>& Time);

```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Time

A `time_point` object.

Dur

A `duration` object.

Return Value

The first function returns a `duration` object that has a time interval that is equal to the sum of the intervals of *Left* and *Right*.

The second and third functions return a `time_point` object that represents a point in time that is displaced, by the interval *Dur*, from the point in time *Time*.

operator<

Determines whether one `duration` or `time_point` object is less than another `duration` or `time_point` object.

```

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Duration2>
constexpr bool operator<(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);

```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

The first function returns **true** if the interval length of *Left* is less than the interval length of *Right*. Otherwise, the function returns **false**.

The second function returns **true** if *Left* precedes *Right*. Otherwise, the function returns **false**.

operator<=

Determines whether one `duration` or `time_point` object is less than or equal to another `duration` or `time_point` object.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Duration2>
constexpr bool operator<=(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

Each function returns `!(Right < Left)`.

operator==

Determines whether two `duration` objects represent time intervals that have the same length, or whether two `time_point` objects represent the same point in time.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

template <class Clock, class Duration1, class Duration2>
constexpr bool operator==(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

The first function returns **true** if *Left* and *Right* represent time intervals that have the same length. Otherwise, the function returns **false**.

The second function returns **true** if *Left* and *Right* represent the same point in time. Otherwise, the function returns **false**.

operator>

Determines whether one [duration](#) or [time_point](#) object is greater than another `duration` or `time_point` object.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(  
    const duration<Rep1, Period1>& Left,  
    const duration<Rep2, Period2>& Right);  
  
template <class Clock, class Duration1, class Duration2>  
constexpr bool operator>(  
    const time_point<Clock, Duration1>& Left,  
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

Each function returns `Right < Left`.

operator>=

Determines whether one [duration](#) or [time_point](#) object is greater than or equal to another `duration` or `time_point` object.

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(  
    const duration<Rep1, Period1>& Left,  
    const duration<Rep2, Period2>& Right);  
  
template <class Clock, class Duration1, class Duration2>  
constexpr bool operator>=(  
    const time_point<Clock, Duration1>& Left,  
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return Value

Each function returns `!(Left < Right)`.

operator modulo

Operator for modulo operations on [duration](#) objects.


```

template <class Rep1, class Period1, class Rep2>
constexpr duration<Rep1, Period1, Rep2>::type
operator%(
    const duration<Rep1, Period1>& Dur,
    const Rep2& Div);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, _Period1>, duration<Rep2, Period2>>::type
operator%(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

```

Parameters

Dur

A `duration` object.

Div

An integral value.

Left

The left `duration` object.

Right

The right `duration` object.

Return Value

The first function returns a `duration` object whose interval length is *Dur* modulo *Div*.

The second function returns a value that represents *Left* modulo *Right*.

See also

[<chrono>](#)

chrono literals

10/31/2018 • 2 minutes to read • [Edit Online](#)

(C++14) The `<chrono>` header defines 12 [user-defined literals](#) to facilitate using literals that represent hours, minutes, seconds, milliseconds, microseconds, and nanoseconds. Each user-defined literal has an integral and a floating-point overload. The literals are defined in the `literals::chrono_literals` inline namespace which is brought into scope automatically when `std::chrono` is in scope.

Syntax

```
inline namespace literals {
    inline namespace chrono_literals {
        // return integral hours
        constexpr chrono::hours operator"" h(unsigned long long Val);

        // return floating-point hours
        constexpr chrono::duration<double, ratio<3600>> operator"" h(long double Val);

        // return integral minutes
        constexpr chrono::minutes(operator"" min)(unsigned long long Val);

        // return floating-point minutes
        constexpr chrono::duration<double, ratio<60>>(operator"" min)(long double Val);

        // return integral seconds
        constexpr chrono::seconds operator"" s(unsigned long long Val);

        // return floating-point seconds
        constexpr chrono::duration<double> operator"" s(long double Val);

        // return integral milliseconds
        constexpr chrono::milliseconds operator"" ms(unsigned long long Val);

        // return floating-point milliseconds
        constexpr chrono::duration<double, milli> operator"" ms(long double Val);

        // return integral microseconds
        constexpr chrono::microseconds operator"" us(unsigned long long Val);

        // return floating-point microseconds
        inline constexpr chrono::duration<double, micro> operator"" us(long double Val);

        // return integral nanoseconds
        inline constexpr chrono::nanoseconds operator"" ns(unsigned long long Val);

        // return floating-point nanoseconds
        constexpr chrono::duration<double, nano> operator"" ns(long double Val);

    } // inline namespace chrono_literals
} // inline namespace literals
```

Return Value

The literals that take a **long long** argument return a value of the corresponding type. The literals that take a floating point argument return a [duration](#).

Example

The following examples show how to use the chrono literals.

```
constexpr auto day = 24h;  
constexpr auto week = 24h* 7;  
constexpr auto my_duration_unit = 108ms;
```

Requirements

Header: <chrono>

Namespace: std::literals::chrono_literals

See also

[<chrono>](#)

common_type Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes specializations of template class [common_type](#) for instantiations of [duration](#) and [time_point](#).

Syntax

```
template <class Rep1, class Period1,
          class Rep2, class Period2>
struct common_type
<chrono::duration<Rep1, Period1>,
 chrono::duration<Rep2, Period2>>;

template <class Clock,
          class Duration1, class Duration2>
struct common_type
<chrono::time_point<Clock, Duration1>,
 chrono::time_point<Clock, Duration2>>;
```

Requirements

Header: <chrono>

Namespace: std

See also

[Header Files Reference](#)

[<chrono>](#)

duration Class

3/28/2019 • 5 minutes to read • [Edit Online](#)

Describes a type that holds a *time interval*, which is an elapsed time between two time points.

Syntax

```
template <class Rep, class Period = ratio<1>>
class duration;
template <class Rep, class Period>
class duration;
template <class Rep, class Period1, class Period2>
class duration <duration<Rep, Period1>, Period2>;
```

Remarks

The template argument `Rep` describes the type that is used to hold the number of clock ticks in the interval. The template argument `Period` is an instantiation of `ratio` that describes the size of the interval that each tick represents.

Members

Public Typedefs

NAME	DESCRIPTION
<code>duration::period</code> Typedef	Represents a synonym for the template parameter <code>Period</code> .
<code>duration::rep</code> Typedef	Represents a synonym for the template parameter <code>Rep</code> .

Public Constructors

NAME	DESCRIPTION
<code>duration</code>	Constructs a <code>duration</code> object.

Public Methods

NAME	DESCRIPTION
<code>count</code>	Returns the number of clock ticks in the time interval.
<code>max</code>	Static. Returns the maximum allowable value of template parameter <code>Rep</code> .
<code>min</code>	Static. Returns the lowest allowable value of template parameter <code>Rep</code> .
<code>zero</code>	Static. In effect, returns <code>Rep(0)</code> .

NAME	DESCRIPTION
------	-------------

Public Operators

NAME	DESCRIPTION
<code>duration::operator-</code>	Returns a copy of the <code>duration</code> object together with a negated tick count.
<code>duration::operator--</code>	Decrements the stored tick count.
<code>duration::operator=</code>	Reduces the stored tick count modulo a specified value.
<code>duration::operator* =</code>	Multiplies the stored tick count by a specified value.
<code>duration::operator/=</code>	Divides the stored tick count by the tick count of a specified <code>duration</code> object.
<code>duration::operator+</code>	Returns <code>*this</code> .
<code>duration::operator++</code>	Increments the stored tick count.
<code>duration::operator+=</code>	Adds the tick count of a specified <code>duration</code> object to the stored tick count.
<code>duration::operator-=</code>	Subtracts the tick count of a specified <code>duration</code> object from the stored tick count.

Requirements

Header: <chrono>

Namespace: std::chrono

duration::count

Retrieves the number of clock ticks in the time interval.

```
constexpr Rep count() const;
```

Return Value

The number of clock ticks in the time interval.

duration::duration Constructor

Constructs a `duration` object.

```
constexpr duration() = default;

template <class Rep2>
constexpr explicit duration(const Rep2& R);

template <class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& Dur);
```

Parameters

Rep2

An arithmetic type to represent the number of ticks.

Period2

A `std::ratio` template specialization to represent the tick period in units of seconds.

R

The number of ticks of default period.

Dur

The number of ticks of period specified by *Period2*.

Remarks

The default constructor constructs an object that is uninitialized. Value initialization by using empty braces initializes an object that represents a time interval of zero clock ticks.

The second, one template argument constructor constructs an object that represents a time interval of *R* clock ticks using a default period of `std::ratio<1>`. To avoid round-off of tick counts, it is an error to construct a duration object from a representation type *Rep2* that can be treated as a floating-point type when `duration::rep` cannot be treated as a floating-point type.

The third, two template argument constructor constructs an object that represents a time interval whose length is the time interval that is specified by *Dur*. To avoid truncation of tick counts, it is an error to construct a duration object from another duration object whose type is *incommensurable* with the target type.

A duration type `D1` is *incommensurable* with another duration type `D2` if `D2` cannot be treated as a floating-point type and `ratio_divide<D1::period, D2::period>::type::den` is not 1.

Unless *Rep2* is implicitly convertible to `rep` and either `treat_as_floating_point<rep>` *holds true* or `treat_as_floating_point<Rep2>` *holds false*, the second constructor does not participate in overload resolution. For more information, see [<type_traits>](#).

Unless no overflow is induced in the conversion and `treat_as_floating_point<rep>` *holds true*, or both `ratio_divide<Period2, period>::den` equals 1 and `treat_as_floating_point<Rep2>` *holds false*, the third constructor does not participate in overload resolution. For more information, see [<type_traits>](#).

duration::max

Static method that returns the upper bound for values of template parameter type `Ref`.

```
static constexpr duration max();
```

Return Value

In effect, returns `duration(duration_values<rep>::max())`.

duration::min

Static method that returns the lower bound for values of template parameter type `Rep`.

```
static constexpr duration min();
```

Return Value

In effect, returns `duration(duration_values<rep>::min())`.

duration::operator-

Returns a copy of the `duration` object together with a negated tick count.

```
constexpr duration operator-() const;
```

duration::operator--

Decrements the stored tick count.

```
duration& operator--();  
  
duration operator--(int);
```

Return Value

The first method returns `*this`.

The second method returns a copy of `*this` that is made before the decrement.

duration::operator=

Reduces the stored tick count modulo a specified value.

```
duration& operator%=(const rep& Div);  
  
duration& operator%=(const duration& Div);
```

Parameters

Div

For the first method, *Div* represents a tick count. For the second method, *Div* is a `duration` object that contains a tick count.

Return Value

The `duration` object after the modulo operation is performed.

duration::operator*=

Multiplies the stored tick count by a specified value.

```
duration& operator*=(const rep& Mult);
```

Parameters

Mult

A value of the type that is specified by `duration::rep`.

Return Value

The `duration` object after the multiplication is performed.

`duration::operator/=`

Divides the stored tick count by a specified value.

```
duration& operator/=(const rep& Div);
```

Parameters

Div

A value of the type that is specified by `duration::rep`.

Return Value

The `duration` object after the division is performed.

`duration::operator+`

Returns `*this`.

```
constexpr duration operator+() const;
```

`duration::operator++`

Increments the stored tick count.

```
duration& operator++();  
  
duration operator++(int);
```

Return Value

The first method returns `*this`.

The second method returns a copy of `*this` that is made before the increment.

`duration::operator+=`

Adds the tick count of a specified `duration` object to the stored tick count.

```
duration& operator+=(const duration& Dur);
```

Parameters

Dur

A `duration` object.

Return Value

The `duration` object after the addition is performed.

`duration::operator-=`

Subtracts the tick count of a specified `duration` object from the stored tick count.

```
duration& operator-=(const duration& Dur);
```

Parameters

Dur

A `duration` object.

Return Value

The `duration` object after the subtraction is performed.

duration::zero

Returns `duration(duration_values<rep>::zero())`.

```
static constexpr duration zero();
```

duration::operator mod=

Reduces the stored tick count modulo `Div` or `Div.count()`.

```
duration& operator%=(const rep& Div);duration& operator%=(const duration& Div);
```

Parameters

Div

The divisor, which is either a duration object or a value that represents tick counts.

Remarks

The first member function reduces the stored tick count modulo `Div` and returns `*this`. The second member function reduces the stored tick count modulo `Div.count()` and returns `*this`.

See also

[Header Files Reference](#)

[<chrono>](#)

[duration_values Structure](#)

duration_values Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Provides specific values for the [duration](#) template parameter `Rep`.

Syntax

```
template <class Rep>
struct duration_values;
```

Members

Public Methods

NAME	DESCRIPTION
max	Static. Specifies the upper limit for a value of type <code>Rep</code> .
min	Static. Specifies the lower limit for a value of type <code>Rep</code> .
zero	Static. Returns <code>Rep(0)</code> .

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

duration_values::max

Static method that returns the upper bound for values of type `Ref`.

```
static constexpr Rep max();
```

Return Value

In effect, returns `numeric_limits<Rep>::max()`.

Remarks

When `Rep` is a user-defined type, the return value must be greater than [duration_values::zero](#).

duration_values::min

Static method that returns the lower bound for values of type `Ref`.

```
static constexpr Rep min();
```

Return Value

In effect, returns `numeric_limits<Rep>::lowest()` .

Remarks

When `Rep` is a user-defined type, the return value must be less than or equal to [duration_values::zero](#).

duration_values::zero

Returns `Rep(0)` .

```
static constexpr Rep zero();
```

Remarks

When `Rep` is a user-defined type, the return value must represent the additive infinity.

See also

[Header Files Reference](#)

[<chrono>](#)

steady_clock struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *steady* clock.

Syntax

```
struct steady_clock;
```

Remarks

On Windows, `steady_clock` wraps the `QueryPerformanceCounter` function.

A clock is *monotonic* if the value that is returned by a first call to `now` is always less than or equal to the value that is returned by a subsequent call to `now`. A clock is *steady* if it is *monotonic* and if the time between clock ticks is constant.

`high_resolution_clock` is a typedef for `steady_clock`.

Public typedefs

NAME	DESCRIPTION
<code>steady_clock::duration</code>	A synonym for <code>nanoseconds</code> , defined in <code><chrono></code> .
<code>steady_clock::period</code>	A synonym for <code>nano</code> , defined in <code><ratio></code> .
<code>steady_clock::rep</code>	A synonym for long long , the type that is used to represent the number of clock ticks in the contained instantiation of <code>duration</code> .
<code>steady_clock::time_point</code>	A synonym for <code>chrono::time_point<steady_clock></code> .

Public functions

FUNCTION	DESCRIPTION
<code>now</code>	Returns the current time as a <code>time_point</code> value.

Public constants

NAME	DESCRIPTION
<code>steady_clock::is_steady</code>	Holds true . A <code>steady_clock</code> is <i>steady</i> .

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

See also

- [Header Files Reference](#)
- [<chrono>](#)
- [system_clock Structure](#)

system_clock Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *clock type* that is based on the real-time clock of the system.

Syntax

```
struct system_clock;
```

Remarks

A *clock type* is used to obtain the current time as UTC. The type embodies an instantiation of [duration](#) and the class template [time_point](#), and defines a static member function `now()` that returns the time.

A clock is *monotonic* if the value that is returned by a first call to `now()` is always less than or equal to the value that is returned by a subsequent call to `now()`.

A clock is *steady* if it is *monotonic* and if the time between clock ticks is constant.

Members

Public Typedefs

NAME	DESCRIPTION
<code>system_clock::duration</code>	A synonym for <code>duration<rep, period></code> .
<code>system_clock::period</code>	A synonym for the type that is used to represent the tick period in the contained instantiation of <code>duration</code> .
<code>system_clock::rep</code>	A synonym for the type that is used to represent the number of clock ticks in the contained instantiation of <code>duration</code> .
<code>system_clock::time_point</code>	A synonym for <code>time_point<Clock, duration></code> , where <code>Clock</code> is a synonym for either the clock type itself or another clock type that is based on the same epoch and has the same nested <code>duration</code> type.

Public Methods

NAME	DESCRIPTION
<code>from_time_t</code>	Static. Returns a <code>time_point</code> that most closely approximates a specified time.
<code>now</code>	Static. Returns the current time.
<code>to_time_t</code>	Static. Returns a <code>time_t</code> object that most closely approximates a specified <code>time_point</code> .

Public Constants

NAME	DESCRIPTION
system_clock::is_monotonic Constant	Specifies whether the clock type is monotonic.
system_clock::is_steady Constant	Specifies whether the clock type is steady.

Requirements

Header: <chrono>

Namespace: std::chrono

system_clock::from_time_t

Static method that returns a [time_point](#) that most closely approximates the time that is represented by *Tm*.

```
static time_point from_time_t(time_t Tm) noexcept;
```

Parameters

Tm

A [time_t](#) object.

system_clock::is_monotonic Constant

Static value that specifies whether the clock type is monotonic.

```
static const bool is_monotonic = false;
```

Return Value

In this implementation, `system_clock::is_monotonic` always returns **false**.

Remarks

A clock is *monotonic* if the value that is returned by a first call to `now()` is always less than or equal to the value that is returned by a subsequent call to `now()`.

system_clock::is_steady Constant

Static value that specifies whether the clock type is *steady*.

```
static const bool is_steady = false;
```

Return Value

In this implementation, `system_clock::is_steady` always returns **false**.

Remarks

A clock is *steady* if it is [monotonic](#) and if the time between clock ticks is constant.

system_clock::now

Static method that returns the current time.


```
static time_point now() noexcept;
```

Return Value

A [time_point](#) object that represents the current time.

system_clock::to_time_t

Static method that returns a [time_t](#) that most closely approximates the time that is represented by *Time*.

```
static time_t to_time_t(const time_point& Time) noexcept;
```

Parameters

Time

A [time_point](#) object.

See also

[Header Files Reference](#)

[<chrono>](#)

[steady_clock](#) struct

time_point Class

3/28/2019 • 2 minutes to read • [Edit Online](#)

A `time_point` describes a type that represents a point in time. It holds an object of type `duration` that stores the elapsed time since the epoch that is represented by the template argument `clock`.

Syntax

```
template <class Clock,
          class Duration = typename Clock::duration>
class time_point;
```

Members

Public Typedefs

NAME	DESCRIPTION
<code>time_point::clock</code>	Synonym for the template parameter <code>Clock</code> .
<code>time_point::duration</code>	Synonym for the template parameter <code>Duration</code> .
<code>time_point::period</code>	Synonym for the nested type name <code>duration::period</code> .
<code>time_point::rep</code>	Synonym for the nested type name <code>duration::rep</code> .

Public Constructors

NAME	DESCRIPTION
<code>time_point</code>	Constructs a <code>time_point</code> object.

Public Methods

NAME	DESCRIPTION
<code>max</code>	Specifies the upper limit for <code>time_point::ref</code> .
<code>min</code>	Specifies the lower limit for <code>time_point::ref</code> .
<code>time_since_epoch</code>	Returns the stored <code>duration</code> value.

Public Operators

NAME	DESCRIPTION
<code>time_point::operator+=</code>	Adds a specified value to the stored duration.

NAME	DESCRIPTION
<code>time_point::operator-=</code>	Subtracts a specified value from the stored duration.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

`time_point::max`

Static method that returns the upper bound for values of type `time_point::ref`.

```
static constexpr time_point max();
```

Return Value

In effect, returns `time_point(duration::max())`.

`time_point::min`

Static method that returns the lower bound for values of type `time_point::ref`.

```
static constexpr time_point min();
```

Return Value

In effect, returns `time_point(duration::min())`.

`time_point::operator+=`

Adds a specified value to the stored [duration](#) value.

```
time_point& operator+=(const duration& Dur);
```

Parameters

Dur

A `duration` object.

Return Value

The `time_point` object after the addition is performed.

`time_point::operator-=`

Subtracts a specified value from the stored [duration](#) value.

```
time_point& operator-=(const duration& Dur);
```

Parameters

Dur

A `duration` object.

Return Value

The `time_point` object after the subtraction is performed.

`time_point::time_point` Constructor

Constructs a `time_point` object.

```
constexpr time_point();

constexpr explicit time_point(const duration& Dur);

template <class Duration2>
constexpr time_point(const time_point<clock, Duration2>& Tp);
```

Parameters

Dur

A [duration](#) object.

Tp

A `time_point` object.

Remarks

The first constructor constructs an object whose stored `duration` value is equal to [duration::zero](#).

The second constructor constructs an object whose stored duration value is equal to *Dur*. Unless `is_convertible<Duration2, duration>` holds true, the second constructor does not participate in overload resolution. For more information, see [<type_traits>](#).

The third constructor initializes its `duration` value by using `Tp.time_since_epoch()`.

`time_point::time_since_epoch`

Retrieves the stored [duration](#) value.

```
constexpr duration time_since_epoch() const;
```

See also

[Header Files Reference](#)

[<chrono>](#)

treat_as_floating_point Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Specifies whether `Rep` can be treated as a floating-point type.

Syntax

```
template <class Rep>
struct treat_as_floating_point : is_floating_point<Rep>;
```

Remarks

`Rep` can be treated as a floating-point type only when the specialization `treat_as_floating_point<Rep>` is derived from [true_type](#). The template class can be specialized for a user-defined type.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

See also

[Header Files Reference](#)

[<chrono>](#)

<inttypes>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <inttypes.h>. Including this header also includes <stdint>.

Syntax

```
#include <inttypes>
```

Remarks

Including this header ensures that the names declared by using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[<stdint>](#)

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<ciso646>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <iso646.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <ciso646>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<climits>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <limits.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <climits>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<locale>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <locale.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <locale>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cmath>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <math.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cmath>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<codecvt>

1/2/2019 • 2 minutes to read • [Edit Online](#)

Defines several template classes that describe objects based on template class `codecvt`. These objects can serve as [locale facets](#) that control conversions between a sequence of values of type `Elem` and a sequence of values of type `char`.

Syntax

```
#include <codecvt>
```

Remarks

The locale facets declared in this header convert between several character encodings. For wide characters (stored within the program in fixed-size integers):

- UCS-4 is Unicode (ISO 10646) encoded within the program as a 32-bit integer.
- UCS-2 is Unicode encoded within the program as a 16-bit integer.
- UTF-16 is Unicode encoded within the program as either one or two 16-bit integers. (Note that this does not meet all the requirements of a valid wide-character encoding for Standard C or Standard C++. Nevertheless it is widely used as such.)

For byte streams (stored in a file, transmitted as a byte sequence, or stored within the program in an array of `char`):

- UTF-8 is Unicode encoded within a byte stream as one or more eight-bit bytes with a deterministic byte order.
- UTF-16LE is Unicode encoded within a byte stream as UTF-16 with each 16-bit integer presented as two eight-bit bytes, less significant byte first.
- UTF-16BE is Unicode encoded within a byte stream as UTF-16 with each 16-bit integer presented as two eight-bit bytes, more significant byte first.

Enumerations

<code>codecvt_mode</code>	Specifies configuration information for locale facets.

Classes

CLASS	DESCRIPTION
<code>codecvt_utf8</code>	Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4, and a byte stream encoded as UTF-8.
<code>codecvt_utf8_utf16</code>	Represents a locale facet that converts between wide characters encoded as UTF-16 and a byte stream encoded as UTF-8.

CLASS	DESCRIPTION
codecvt_utf16	Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE or UTF-16BE.

Requirements

Header: <codecvt>

Namespace: std

See also

[Header Files Reference](#)

<codecvt> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

codecvt_mode Enumeration

Specifies configuration information for [locale](#) facets.

```
enum codecvt_mode {  
    consume_header = 4,  
    generate_header = 2,  
    little_endian = 1  
};
```

Remarks

The enumeration defines three constants that supply configuration information to the locale facets declared in [<codecvt>](#). The distinct values are:

- `consume_header`, to consume an initial header sequence when reading a multibyte sequence and determine the endianness of the subsequent multibyte sequence to be read
- `generate_header`, to generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written
- `little_endian`, to generate a multibyte sequence in little-endian order, as opposed to the default big-endian order

These constants can be ORed together in arbitrary combinations.

See also

[<codecvt>](#)

codecvt_utf8

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a [locale](#) facet that converts between wide characters encoded as UCS-2 or UCS-4, and a byte stream encoded as UTF-8.

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
class codecvt_utf8 : public std::codecvt<Elem, char, StateType>
```

Parameters

Elem

The wide-character element type.

Maxcode

The maximum number of characters for the locale facet.

Mode

Configuration information for the locale facet.

Remarks

The byte stream can be written to either a binary file or a text file.

Requirements

Header: <codecvt>\

Namespace: std

codecvt_utf8_utf16

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a [locale](#) facet that converts between wide characters encoded as UTF-16 and a byte stream encoded as UTF-8.

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
class codecvt_utf8_utf16 : public _STD codecvt<Elem, char, StateType>
```

Parameters

Elem

The wide-character element type.

Maxcode

The maximum number of characters for the locale facet.

Mode

Configuration information for the locale facet.

Remarks

The byte stream can be written to either a binary file or a text file.

Requirements

Header: <codecvt>

Namespace: std

codecvt_utf16

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a [locale](#) facet that converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE or UTF-16BE.

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
class codecvt_utf16 : public std::codecvt<Elem, char, StateType>
```

Parameters

Elem

The wide-character element type.

Maxcode

The maximum number of characters for the locale facet.

Mode

Configuration information for the locale facet.

Remarks

This template class converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE, if Mode & little_endian, or UTF-16BE otherwise.

The byte stream should be written to a binary file; it can be corrupted if written to a text file.

Requirements

Header: <codecvt>

Namespace: std

<complex>

10/31/2018 • 5 minutes to read • [Edit Online](#)

Defines the container template class `complex` and its supporting templates.

Syntax

```
#include <complex>
```

Remarks

A complex number is an ordered pair of real numbers. In purely geometrical terms, the complex plane is the real, two-dimensional plane. The special qualities of the complex plane that distinguish it from the real plane are due to its having an additional algebraic structure. This algebraic structure has two fundamental operations:

- Addition defined as $(a, b) + (c, d) = (a + c, b + d)$
- Multiplication defined as $(a, b) * (c, d) = (ac - bd, ad + bc)$

The set of complex numbers with the operations of complex addition and complex multiplication are a field in the standard algebraic sense:

- The operations of addition and multiplication are commutative and associative and multiplication distributes over addition exactly as it does with real addition and multiplication on the field of real numbers.
- The complex number $(0, 0)$ is the additive identity and $(1, 0)$ is the multiplicative identity.
- The additive inverse for a complex number (a, b) is $(-a, -b)$, and the multiplicative inverse for all such complex numbers except $(0, 0)$ is

$$(a/(a^2 + b^2), -b/(a^2 + b^2))$$

By representing a complex number $z = (a, b)$ in the form $z = a + bi$, where $i^2 = -1$, the rules for the algebra of the set of real numbers can be applied to the set of complex numbers and to their components. For example:

$$(1 + 2i) * (2 + 3i) = 1 * (2 + 3i) + 2i * (2 + 3i) = (2 + 3i) + (4i + 6i^2) = (2 - 6) + (3 + 4)i = -4 + 7i$$

The system of complex numbers is a field, but it is not an ordered field. There is no ordering of the complex numbers as there is for the field of real numbers and its subsets, so inequalities cannot be applied to complex numbers as they are to real numbers.

There are three common forms of representing a complex number z :

- Cartesian: $z = a + bi$
- Polar: $z = r (\cos p + i \sin p)$
- Exponential: $z = r * e^{ip}$

The terms used in these standard representations of a complex number are referred to as follows:

- The real Cartesian component or real part a .
- The imaginary Cartesian component or imaginary part b .

- The modulus or absolute value of a complex number r .
- The argument or phase angle p in radians.

Unless otherwise specified, functions that can return multiple values are required to return a principal value for their arguments greater than $-\pi$ and less than or equal to $+\pi$ to keep them single valued. All angles must be expressed in radians, where there are 2π radians (360 degrees) in a circle.

Functions

FUNCTION	DESCRIPTION
<code>abs</code>	Calculates the modulus of a complex number.
<code>arg</code>	Extracts the argument from a complex number.
<code>conj</code>	Returns the complex conjugate of a complex number.
<code>cos</code>	Returns the cosine of a complex number.
<code>cosh</code>	Returns the hyperbolic cosine of a complex number.
<code>exp</code>	Returns the exponential function of a complex number.
<code>imag</code>	Extracts the imaginary component of a complex number.
<code>log</code>	Returns the natural logarithm of a complex number.
<code>log10</code>	Returns the base 10 logarithm of a complex number.
<code>norm</code>	Extracts the norm of a complex number.
<code>polar</code>	Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.
<code>pow</code>	Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.
<code>real</code>	Extracts the real component of a complex number.
<code>sin</code>	Returns the sine of a complex number.
<code>sinh</code>	Returns the hyperbolic sine of a complex number.
<code>sqrt</code>	Returns the square root of a complex number.
<code>tan</code>	Returns the tangent of a complex number.
<code>tanh</code>	Returns the hyperbolic tangent of a complex number.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator*</code>	Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator+</code>	Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator-</code>	Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator/</code>	Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator<<</code>	A template function that inserts a complex number into the output stream.
<code>operator==</code>	Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
<code>operator>></code>	A template function that extracts a complex value from the input stream.

Classes

CLASS	DESCRIPTION
<code>complex<double></code>	The explicitly specialized template class describes an object that stores an ordered pair of objects, both of type double , where the first represents the real part of a complex number and the second represents the imaginary part.
<code>complex<float></code>	The explicitly specialized template class describes an object that stores an ordered pair of objects, both of type float , where the first represents the real part of a complex number and the second represents the imaginary part.
<code>complex<long double></code>	The explicitly specialized template class describes an object that stores an ordered pair of objects, both of type long double , where the first represents the real part of a complex number and the second represents the imaginary part.
<code>complex</code>	The template class describes an object used to represent the complex number system and perform complex arithmetic operations.

Literals

The `<complex>` header defines the following [user-defined literals](#) which create a complex number with the real

part being zero and the imaginary part being the value of the input parameter.

<div>constexpr complex<long double> operator""il(long double d)</div> <div>constexpr complex<long double> operator""il(unsigned long long d)</div>	<div>Returns:</div> <div>complex<long double>{0.0L, static_cast<long double>(d)}</div>
<div>constexpr complex<double> operator""i(long double d)</div> <div>constexpr complex<double> operator""i(unsigned long long d)</div>	<div>Returns:</div> <div>complex<double>{0.0, static_cast<double>(d)} .</div>
<div>constexpr complex<float> operator""if(long double d)</div> <div>constexpr complex<float> operator""if(unsigned long long d)</div>	<div>Returns:</div> <div>complex<float>{0.0f, static_cast<float>(d)} .</div>

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<complex> functions

10/31/2018 • 36 minutes to read • [Edit Online](#)

abs	arg	conj
cos	cosh	exp
imag	log	log10
norm	polar	pow
real	sin	sinh
sqrt	tan	tanh

abs

Calculates the modulus of a complex number.

```
template <class Type>
Type abs(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose modulus is to be determined.

Return Value

The modulus of a complex number.

Remarks

The *modulus* of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number $a + bi$ is $\sqrt{a^2 + b^2}$, written $|a + bi|$. The *norm* of a complex number $a + bi$ is $(a^2 + b^2)$, so the modulus of a complex number is the square root of its norm.

Example

```

// complex_abs.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Complex numbers can be entered in polar form with
    // modulus and argument parameter inputs but are
    // stored in Cartesian form as real & imag coordinates
    complex <double> c1 ( polar ( 5.0 ) ); // Default argument = 0
    complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
    complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
    cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
    cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
    cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

    // The modulus and argument of a complex number can be recovered
    // using abs & arg member functions
    double absc1 = abs ( c1 );
    double argc1 = arg ( c1 );
    cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
    cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

    // Testing if the principal angles of c2 and c3 are the same
    if ( ( arg ( c2 ) <= ( arg ( c3 ) + .00000001 ) ) ||
        ( arg ( c2 ) >= ( arg ( c3 ) - .00000001 ) ) )
        cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
        cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;
}

```

```

c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.

```

arg

Extracts the argument from a complex number.

```
template <class Type>
Type arg(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose argument is to be determined.

Return Value

The argument of the complex number.

Remarks

The *argument* is the angle that the complex vector makes with the positive real axis in the complex plane. For a complex number $a + bi$, the argument is equal to $\arctan(b/a)$. The angle has a positive sense when measured in a counterclockwise direction from the positive real axis and a negative sense when measured in a clockwise direction. The principal values are greater than $-\pi$ and less than or equal to $+\pi$.

Example

```

// complex_arg.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Complex numbers can be entered in polar form with
    // modulus and argument parameter inputs but are
    // stored in Cartesian form as real & imag coordinates
    complex <double> c1 ( polar ( 5.0 ) ); // Default argument = 0
    complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
    complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
    cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
    cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
    cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

    // The modulus and argument of a complex number can be recovered
    // using abs & arg member functions
    double absc1 = abs ( c1 );
    double argc1 = arg ( c1 );
    cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
    cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

    // Testing if the principal angles of c2 and c3 are the same
    if ( ( arg ( c2 ) <= ( arg ( c3 ) + .00000001 ) ) ||
        ( arg ( c2 ) >= ( arg ( c3 ) - .00000001 ) ) )
        cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
        cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;
}

```

```

c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.

```

conj

Returns the complex conjugate of a complex number.


```
template <class Type>
complex<Type> conj(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose complex conjugate is being returned.

Return Value

The complex conjugate of the input complex number.

Remarks

The complex conjugate of a complex number $a + bi$ is $a - bi$. The product of a complex number and its conjugate is the norm of the number $a^2 + b^2$.

Example

```
// complex_conj.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;

    complex <double> c1 ( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = real ( c1 );
    cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

    double di1 = imag ( c1 );
    cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;

    complex <double> c2 = conj ( c1 );
    cout << "The complex conjugate of c1 is c2 = conj ( c1 ) = "
        << c2 << endl;

    double dr2 = real ( c2 );
    cout << "The real part of c2 is real ( c2 ) = "
        << dr2 << "." << endl;

    double di2 = imag ( c2 );
    cout << "The imaginary part of c2 is imag ( c2 ) = "
        << di2 << "." << endl;

    // The real part of the product of a complex number
    // and its conjugate is the norm of the number
    complex <double> c3 = c1 * c2;
    cout << "The norm of (c1 * conj (c1) ) is c1 * c2 = "
        << real( c3 ) << endl;
}
```

The complex number $c1 = (4,3)$
The real part of $c1$ is $\text{real} (c1) = 4$.
The imaginary part of $c1$ is $\text{imag} (c1) = 3$.
The complex conjugate of $c1$ is $c2 = \text{conj} (c1) = (4,-3)$
The real part of $c2$ is $\text{real} (c2) = 4$.
The imaginary part of $c2$ is $\text{imag} (c2) = -3$.
The norm of $(c1 * \text{conj} (c1))$ is $c1 * c2 = 25$

COS

Returns the cosine of a complex number.

```
template <class Type>
complex<Type> cos(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose cosine is being determined.

Return Value

The complex number that is the cosine of the input complex number.

Remarks

Identities defining the complex cosines:

$$\cos (z) = (1/2) * (\exp (iz) + \exp (- iz))$$

$$\cos (z) = \cos (a + bi) = \cos (a) \cosh (b) - i \sin (a) \sinh (b)$$

Example

```

// complex_cos.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of cosine of a complex number c1
    complex <double> c2 = cos ( c1 );
    cout << "Complex number c2 = cos ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Cosines of the standard angles in the first
    // two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar (1.0, pi / 6) );
    v1.push_back( cos ( vc1 ) );
    complex <double> vc2 ( polar (1.0, pi / 3) );
    v1.push_back( cos ( vc2 ) );
    complex <double> vc3 ( polar (1.0, pi / 2) );
    v1.push_back( cos ( vc3 ) );
    complex <double> vc4 ( polar (1.0, 2 * pi / 3) );
    v1.push_back( cos ( vc4 ) );
    complex <double> vc5 ( polar (1.0, 5 * pi / 6) );
    v1.push_back( cos ( vc5 ) );
    complex <double> vc6 ( polar (1.0, pi) );
    v1.push_back( cos ( vc6 ) );

    cout << "The complex components cos (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}

```

```

Complex number c1 = (3,4)
Complex number c2 = cos ( c1 ) = (-27.0349,-3.85115)
The modulus of c2 is: 27.3079
The argument of c2 is: -3.00009 radians, which is -171.893 degrees.

The complex components cos (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,1.21529e-013)
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-1.74036e-013)

```

cosh

Returns the hyperbolic cosine of a complex number.

```
template <class Type>
complex<Type> cosh(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose hyperbolic cosine is being determined.

Return Value

The complex number that is the hyperbolic cosine of the input complex number.

Remarks

Identities defining the complex hyperbolic cosines:

$$\cos (z) = (1 / 2) * (\exp (z) + \exp (- z))$$

$$\cos (z) = \cosh (a + bi) = \cosh (a) \cos (b) + i \sinh (a) \sin (b)$$

Example

```

// complex_cosh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of cosine of a complex number c1
    complex <double> c2 = cosh ( c1 );
    cout << "Complex number c2 = cosh ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Hyperbolic cosines of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar (1.0, pi / 6) );
    v1.push_back( cosh ( vc1 ) );
    complex <double> vc2 ( polar (1.0, pi / 3) );
    v1.push_back( cosh ( vc2 ) );
    complex <double> vc3 ( polar (1.0, pi / 2) );
    v1.push_back( cosh ( vc3 ) );
    complex <double> vc4 ( polar (1.0, 2 * pi / 3) );
    v1.push_back( cosh ( vc4 ) );
    complex <double> vc5 ( polar (1.0, 5 * pi / 6) );
    v1.push_back( cosh ( vc5 ) );
    complex <double> vc6 ( polar (1.0, pi) );
    v1.push_back( cosh ( vc6 ) );

    cout << "The complex components cosh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}

```

```

Complex number c1 = (3,4)
Complex number c2 = cosh ( c1 ) = (-6.58066,-7.58155)
The modulus of c2 is: 10.0392
The argument of c2 is: -2.28564 radians, which is -130.957 degrees.

The complex components cosh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-8.70178e-014)
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,2.43059e-013)

```

exp

Returns the exponential function of a complex number.

```
template <class Type>
complex<Type> exp(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose exponential is being determined.

Return Value

The complex number that is the exponential of the input complex number.

Example

```
// complex_exp.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 1 , pi/6 );
    cout << "Complex number c1 = " << c1 << endl;

    // Value of exponential of a complex number c1:
    // note the argument of c2 is determined by the
    // imaginary part of c1 & the modulus by the real part
    complex <double> c2 = exp ( c1 );
    cout << "Complex number c2 = exp ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Exponentials of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( 0.0 , -pi );
    v1.push_back( exp ( vc1 ) );
    complex <double> vc2 ( 0.0, -2 * pi / 3 );
    v1.push_back( exp ( vc2 ) );
    complex <double> vc3 ( 0.0, 0.0 );
    v1.push_back( exp ( vc3 ) );
    complex <double> vc4 ( 0.0, pi / 3 );
    v1.push_back( exp ( vc4 ) );
    complex <double> vc5 ( 0.0 , 2 * pi / 3 );
    v1.push_back( exp ( vc5 ) );
    complex <double> vc6 ( 0.0, pi );
    v1.push_back( exp ( vc6 ) );

    cout << "The complex components exp (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 3 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
        cout << ( * Iter1 ) << "\n    with argument = "
            << ( 180/pi ) * arg ( *Iter1 )
            << " degrees\n    modulus = "
            << abs ( * Iter1 ) << endl;
}
```

Extracts the imaginary component of a complex number.

```
template <class Type>
Type imag(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose real part is to be extracted.

Return Value

The imaginary part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

```
// complexc_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    complex <double> c1 ( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = real ( c1 );
    cout << "The real part of c1 is real ( c1 ) = "
         << dr1 << "." << endl;

    double di1 = imag ( c1 );
    cout << "The imaginary part of c1 is imag ( c1 ) = "
         << di1 << "." << endl;
}
```

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

log

Returns the natural logarithm of a complex number.

```
template <class Type>
complex<Type> log(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose natural logarithm is being determined.

Return Value

The complex number that is the natural logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

```
// complex_log.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of log of a complex number c1
    complex <double> c2 = log ( c1 );
    cout << "Complex number c2 = log ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // log of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar (1.0, pi / 6) );
    v1.push_back( log ( vc1 ) );
    complex <double> vc2 ( polar (1.0, pi / 3) );
    v1.push_back( log ( vc2 ) );
    complex <double> vc3 ( polar (1.0, pi / 2) );
    v1.push_back( log ( vc3 ) );
    complex <double> vc4 ( polar (1.0, 2 * pi / 3) );
    v1.push_back( log ( vc4 ) );
    complex <double> vc5 ( polar (1.0, 5 * pi / 6) );
    v1.push_back( log ( vc5 ) );
    complex <double> vc6 ( polar (1.0, pi) );
    v1.push_back( log ( vc6 ) );

    cout << "The complex components log (vci), where abs (vci) = 1 "
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
        cout << *Iter1 << " " << endl;
}
```

log10

Returns the base 10 logarithm of a complex number.

```
template <class Type>
complex<Type> log10(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose base 10 logarithm is being determined.

Return Value

The complex number that is the base 10 logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

```
// complex_log10.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of log10 of a complex number c1
    complex <double> c2 = log10 ( c1 );
    cout << "Complex number c2 = log10 ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // log10 of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar (1.0, pi / 6) );
    v1.push_back( log10 ( vc1 ) );
    complex <double> vc2 ( polar (1.0, pi / 3) );
    v1.push_back( log10 ( vc2 ) );
    complex <double> vc3 ( polar (1.0, pi / 2) );
    v1.push_back( log10 ( vc3 ) );
    complex <double> vc4 ( polar (1.0, 2 * pi / 3) );
    v1.push_back( log10 ( vc4 ) );
    complex <double> vc5 ( polar (1.0, 5 * pi / 6) );
    v1.push_back( log10 ( vc5 ) );
    complex <double> vc6 ( polar (1.0, pi) );
    v1.push_back( log10 ( vc6 ) );

    cout << "The complex components log10 (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}
```

norm

Extracts the norm of a complex number.

```
template <class Type>
Type norm(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose norm is to be determined.

Return Value

The norm of a complex number.

Remarks

The norm of a complex number $a + bi$ is $(a^2 + b^2)$. The norm of a complex number is the square of its modulus. The modulus of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number $a + bi$ is $\sqrt{a^2 + b^2}$, written $|a + bi|$.

Example

```
// complex_norm.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Complex numbers can be entered in polar form with
    // modulus and argument parameter inputs but are
    // stored in Cartesian form as real & imag coordinates
    complex <double> c1 ( polar ( 5.0 ) ); // Default argument = 0
    complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
    complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
    cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
    cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
    cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

    if ( ( arg ( c2 ) <= ( arg ( c3 ) + .00000001 ) ) ||
        ( arg ( c2 ) >= ( arg ( c3 ) - .00000001 ) ) )
        cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
        cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;

    // The modulus and argument of a complex number can be recovered
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

    // The norm of a complex number is the square of its modulus
    double normc2 = norm ( c2 );
    double sqrtnormc2 = sqrt ( normc2 );
    cout << "The norm of c2 given by: norm ( c2 ) = " << normc2 << endl;
    cout << "The modulus of c2 is the square root of the norm: "
        << "sqrt ( normc2 ) = " << sqrtnormc2 << ".";
}
```

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The norm of c2 given by: norm ( c2 ) = 25
The modulus of c2 is the square root of the norm: sqrt ( normc2 ) = 5.
```

polar

Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.

```
template <class Type>
complex<Type> polar(const Type& _Modulus, const Type& _Argument = 0);
```

Parameters

_Modulus

The modulus of the complex number being input.

_Argument

The argument of the complex number being input.

Return Value

Cartesian form of the complex number specified in polar form.

Remarks

The polar form of a complex number provides the modulus r and the argument p , where these parameters are related to the real and imaginary Cartesian components a and b by the equations $a = r * \cos p$ and $b = r * \sin p$.

Example

```
// complex_polar.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Complex numbers can be entered in polar form with
    // modulus and argument parameter inputs but are
    // stored in Cartesian form as real & imag coordinates
    complex <double> c1 ( polar ( 5.0 ) ); // Default argument = 0
    complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
    complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
    cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
    cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
    cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

    if ( ( arg ( c2 ) <= ( arg ( c3 ) + .00000001 ) ) ||
        ( arg ( c2 ) >= ( arg ( c3 ) - .00000001 ) ) )
        cout << "The complex numbers c2 & c3 have the "
            << "same principal arguments."<< endl;
    else
        cout << "The complex numbers c2 & c3 don't have the "
            << "same principal arguments." << endl;

    // the modulus and argument of a complex number can be recovered
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;
}
```

```

c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.

```

pow

Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.

```

template <class Type>
complex<Type> pow(const complex<Type>& _Base, int _Power);

template <class Type>
complex<Type> pow(const complex<Type>& _Base, const Type& _Power);

template <class Type>
complex<Type> pow(const complex<Type>& _Base, const complex<Type>& _Power);

template <class Type>
complex<Type> pow(const Type& _Base, const complex<Type>& _Power);

```

Parameters

_Base

The complex number or number that is of the parameter type for the complex number that is the base to be raised to a power by the member function.

_Power

The integer or complex number or number that is of the parameter type for the complex number that is the power that the base is to be raised to by the member function.

Return Value

The complex number obtained by raising the specified base to the specified power.

Remarks

The functions each effectively convert both operands to the return type, and then return the converted **left** to the power **right**.

The branch cut is along the negative real axis.

Example

```

// complex_pow.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // First member function
    // type complex<double> base & type integer power
    complex <double> cb1 ( 3 , 4);
    int cp1 = 2;
    complex <double> cp1 = pow ( cb1 , cp1 );
}

```

```

complex <double> ce1 = pow ( cb1 ,cp1 );

cout << "Complex number for base cb1 = " << cb1 << endl;
cout << "Integer for power = " << cp1 << endl;
cout << "Complex number returned from complex base and integer power:"
    << "\n ce1 = cb1 ^ cp1 = " << ce1 << endl;
double absce1 = abs ( ce1 );
double argce1 = arg ( ce1 );
cout << "The modulus of ce1 is: " << absce1 << endl;
cout << "The argument of ce1 is: "<< argce1 << " radians, which is "
    << argce1 * 180 / pi << " degrees." << endl << endl;

// Second member function
// type complex<double> base & type double power
complex <double> cb2 ( 3 , 4 );
double cp2 = pi;
complex <double> ce2 = pow ( cb2 ,cp2 );

cout << "Complex number for base cb2 = " << cb2 << endl;
cout << "Type double for power cp2 = pi = " << cp2 << endl;
cout << "Complex number returned from complex base and double power:"
    << "\n ce2 = cb2 ^ cp2 = " << ce2 << endl;
double absce2 = abs ( ce2 );
double argce2 = arg ( ce2 );
cout << "The modulus of ce2 is: " << absce2 << endl;
cout << "The argument of ce2 is: "<< argce2 << " radians, which is "
    << argce2 * 180 / pi << " degrees." << endl << endl;

// Third member function
// type complex<double> base & type complex<double> power
complex <double> cb3 ( 3 , 4 );
complex <double> cp3 ( -2 , 1 );
complex <double> ce3 = pow ( cb3 ,cp3 );

cout << "Complex number for base cb3 = " << cb3 << endl;
cout << "Complex number for power cp3= " << cp3 << endl;
cout << "Complex number returned from complex base and complex power:"
    << "\n ce3 = cb3 ^ cp3 = " << ce3 << endl;
double absce3 = abs ( ce3 );
double argce3 = arg ( ce3 );
cout << "The modulus of ce3 is: " << absce3 << endl;
cout << "The argument of ce3 is: "<< argce3 << " radians, which is "
    << argce3 * 180 / pi << " degrees." << endl << endl;

// Fourth member function
// type double base & type complex<double> power
double cb4 = pi;
complex <double> cp4 ( 2 , -1 );
complex <double> ce4 = pow ( cb4 ,cp4 );

cout << "Type double for base cb4 = pi = " << cb4 << endl;
cout << "Complex number for power cp4 = " << cp4 << endl;
cout << "Complex number returned from double base and complex power:"
    << "\n ce4 = cb4 ^ cp4 = " << ce4 << endl;
double absce4 = abs ( ce4 );
double argce4 = arg ( ce4 );
cout << "The modulus of ce4 is: " << absce4 << endl;
cout << "The argument of ce4 is: "<< argce4 << " radians, which is "
    << argce4 * 180 / pi << " degrees." << endl << endl;
}

```

```

Complex number for base cb1 = (3,4)
Integer for power = 2
Complex number returned from complex base and integer power:
ce1 = cb1 ^ cp1 = (-7,24)
The modulus of ce1 is: 25
The argument of ce1 is: 1.85459 radians, which is 106.26 degrees.

Complex number for base cb2 = (3,4)
Type double for power cp2 = pi = 3.14159
Complex number returned from complex base and double power:
ce2 = cb2 ^ cp2 = (-152.915,35.5475)
The modulus of ce2 is: 156.993
The argument of ce2 is: 2.91318 radians, which is 166.913 degrees.

Complex number for base cb3 = (3,4)
Complex number for power cp3= (-2,1)
Complex number returned from complex base and complex power:
ce3 = cb3 ^ cp3 = (0.0153517,-0.00384077)
The modulus of ce3 is: 0.0158249
The argument of ce3 is: -0.245153 radians, which is -14.0462 degrees.

Type double for base cb4 = pi = 3.14159
Complex number for power cp4 = (2,-1)
Complex number returned from double base and complex power:
ce4 = cb4 ^ cp4 = (4.07903,-8.98725)
The modulus of ce4 is: 9.8696
The argument of ce4 is: -1.14473 radians, which is -65.5882 degrees.

```

real

Extracts the real component of a complex number.

```

template <class Type>
Type real(const complex<Type>& complexNum);

```

Parameters

complexNum

The complex number whose real part is to be extracted.

Return Value

The real part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

```
// complex_real.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    complex <double> c1 ( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = real ( c1 );
    cout << "The real part of c1 is real ( c1 ) = "
         << dr1 << "." << endl;

    double di1 = imag ( c1 );
    cout << "The imaginary part of c1 is imag ( c1 ) = "
         << di1 << "." << endl;
}
```

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

sin

Returns the sine of a complex number.

```
template <class Type>
complex<Type> sin(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose sine is being determined.

Return Value

The complex number that is the sine of the input complex number.

Remarks

Identities defining the complex sines:

$$\sin (z) = (1 / 2 \, i) * (\exp (i z) - \exp (- i z))$$

$$\sin (z) = \sin (a + bi) = \sin (a) \cosh (b) + i \cos (a) \sinh (b)$$

Example

```

// complex_sin.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of sine of a complex number c1
    complex <double> c2 = sin ( c1 );
    cout << "Complex number c2 = sin ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // sines of the standard angles in the first
    // two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar ( 1.0, pi / 6 ) );
    v1.push_back( sin ( vc1 ) );
    complex <double> vc2 ( polar ( 1.0, pi / 3 ) );
    v1.push_back( sin ( vc2 ) );
    complex <double> vc3 ( polar ( 1.0, pi / 2 ) );
    v1.push_back( sin ( vc3 ) );
    complex <double> vc4 ( polar ( 1.0, 2 * pi / 3 ) );
    v1.push_back( sin ( vc4 ) );
    complex <double> vc5 ( polar ( 1.0, 5 * pi / 6 ) );
    v1.push_back( sin ( vc5 ) );
    complex <double> vc6 ( polar ( 1.0, pi ) );
    v1.push_back( sin ( vc6 ) );

    cout << "The complex components sin (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}

```

```

Complex number c1 = (3,4)
Complex number c2 = sin ( c1 ) = (3.85374,-27.0168)
The modulus of c2 is: 27.2903
The argument of c2 is: -1.42911 radians, which is -81.882 degrees.

```

```

The complex components sin (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.85898,0.337596)
(0.670731,0.858637)
(-1.59572e-013,1.1752)
(-0.670731,0.858637)
(-0.85898,0.337596)
(-0.841471,-1.11747e-013)

```

sinh

Returns the hyperbolic sine of a complex number.


```
template <class Type>
complex<Type> sinh(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose hyperbolic sine is being determined.

Return Value

The complex number that is the hyperbolic sine of the input complex number.

Remarks

Identities defining the complex hyperbolic sines:

$$\sinh (z) = (1 / 2) * (\exp (z) - \exp (- z))$$

$$\sinh (z) = \sinh (a + bi) = \sinh (a) \cos (b) + i \cosh (a) \sin (b)$$

Example

```

// complex_sinh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of sine of a complex number c1
    complex <double> c2 = sinh ( c1 );
    cout << "Complex number c2 = sinh ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Hyperbolic sines of the standard angles in
    // the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar ( 1.0, pi / 6 ) );
    v1.push_back( sinh ( vc1 ) );
    complex <double> vc2 ( polar ( 1.0, pi / 3 ) );
    v1.push_back( sinh ( vc2 ) );
    complex <double> vc3 ( polar ( 1.0, pi / 2 ) );
    v1.push_back( sinh ( vc3 ) );
    complex <double> vc4 ( polar ( 1.0, 2 * pi / 3 ) );
    v1.push_back( sinh ( vc4 ) );
    complex <double> vc5 ( polar ( 1.0, 5 * pi / 6 ) );
    v1.push_back( sinh ( vc5 ) );
    complex <double> vc6 ( polar ( 1.0, pi ) );
    v1.push_back( sinh ( vc6 ) );

    cout << "The complex components sinh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}

```

```

Complex number c1 = (3,4)
Complex number c2 = sinh ( c1 ) = (-6.54812,-7.61923)
The modulus of c2 is: 10.0464
The argument of c2 is: -2.28073 radians, which is -130.676 degrees.

The complex components sinh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.858637,0.670731)
(0.337596,0.85898)
(-5.58735e-014,0.841471)
(-0.337596,0.85898)
(-0.858637,0.670731)
(-1.1752,-3.19145e-013)

```

sqrt

Calculates the square root of a complex number.

```
template <class Type>
complex<Type> sqrt(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose square root is to be found.

Return Value

The square root of a complex number.

Remarks

The square root will have a phase angle in the half-open interval $(-\pi/2, \pi/2]$.

The branch cuts in the complex plane are along the negative real axis.

The square root of a complex number will have a modulus that is the square root of the input number and an argument that is one-half that of the input number.

Example

```
// complex_sqrt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Complex numbers can be entered in polar form with
    // modulus and argument parameter inputs but are
    // stored in Cartesian form as real & imag coordinates
    complex <double> c1 ( polar ( 25.0 , pi / 2 ) );
    complex <double> c2 = sqrt ( c1 );
    cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
    cout << "c2 = sqrt ( c1 ) = " << c2 << endl;

    // The modulus and argument of a complex number can be recovered
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

    // The modulus and argument of c2 can be directly calculated
    absc2 = sqrt( abs ( c1 ) );
    argc2 = 0.5 * arg ( c1 );
    cout << "The modulus of c2 = sqrt( abs ( c1 ) ) =" << absc2 << endl;
    cout << "The argument of c2 = ( 1 / 2 ) * arg ( c1 ) ="
        << argc2 << " radians,\n which is " << argc2 * 180 / pi
        << " degrees." << endl;
}
```

```
c1 = polar ( 5.0 ) = (-2.58529e-012,25)
c2 = sqrt ( c1 ) = (3.53553,3.53553)
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.785398 radians, which is 45 degrees.
The modulus of c2 = sqrt( abs ( c1 ) ) =5
The argument of c2 = ( 1 / 2 ) * arg ( c1 ) =0.785398 radians,
which is 45 degrees.
```

tan

Returns the tangent of a complex number.

```
template <class Type>
complex<Type> tan(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose tangent is being determined.

Return Value

The complex number that is the tangent of the input complex number.

Remarks

Identities defining the complex cotangent:

$$\tan (z) = \sin (z) / \cos (z) = (\exp (iz) - \exp (- iz)) / i (\exp (iz) + \exp (- iz))$$

Example

```

// complex_tan.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of cosine of a complex number c1
    complex <double> c2 = tan ( c1 );
    cout << "Complex number c2 = tan ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Hyperbolic tangent of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar ( 1.0, pi / 6 ) );
    v1.push_back( tan ( vc1 ) );
    complex <double> vc2 ( polar ( 1.0, pi / 3 ) );
    v1.push_back( tan ( vc2 ) );
    complex <double> vc3 ( polar ( 1.0, pi / 2 ) );
    v1.push_back( tan ( vc3 ) );
    complex <double> vc4 ( polar ( 1.0, 2 * pi / 3 ) );
    v1.push_back( tan ( vc4 ) );
    complex <double> vc5 ( polar ( 1.0, 5 * pi / 6 ) );
    v1.push_back( tan ( vc5 ) );
    complex <double> vc6 ( polar ( 1.0, pi ) );
    v1.push_back( tan ( vc6 ) );

    cout << "The complex components tan (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
        cout << *Iter1 << endl;
}

```

```

Complex number c1 = (3,4)
Complex number c2 = tan ( c1 ) = (-0.000187346,0.999356)
The modulus of c2 is: 0.999356
The argument of c2 is: 1.57098 radians, which is 90.0107 degrees.

The complex components tan (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.713931,0.85004)
(0.24356,0.792403)
(-4.34302e-014,0.761594)
(-0.24356,0.792403)
(-0.713931,0.85004)
(-1.55741,-7.08476e-013)

```

tanh

Returns the hyperbolic tangent of a complex number.

```
template <class Type>
complex<Type> tanh(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose hyperbolic tangent is being determined.

Return Value

The complex number that is the hyperbolic tangent of the input complex number.

Remarks

Identities defining the complex hyperbolic cotangent:

$$\tanh (z)=\sinh (z) / \cosh (z)=(\exp (z)-\exp (-z)) /(\exp (z)+\exp (-z))$$

Example

```
// complex_tanh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    complex <double> c1 ( 3.0 , 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    // Values of cosine of a complex number c1
    complex <double> c2 = tanh ( c1 );
    cout << "Complex number c2 = tanh ( c1 ) = " << c2 << endl;
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

    // Hyperbolic tangents of the standard angles
    // in the first two quadrants of the complex plane
    vector <complex <double> > v1;
    vector <complex <double> >::iterator Iter1;
    complex <double> vc1 ( polar ( 1.0, pi / 6 ) );
    v1.push_back( tanh ( vc1 ) );
    complex <double> vc2 ( polar ( 1.0, pi / 3 ) );
    v1.push_back( tanh ( vc2 ) );
    complex <double> vc3 ( polar ( 1.0, pi / 2 ) );
    v1.push_back( tanh ( vc3 ) );
    complex <double> vc4 ( polar ( 1.0, 2 * pi / 3 ) );
    v1.push_back( tanh ( vc4 ) );
    complex <double> vc5 ( polar ( 1.0, 5 * pi / 6 ) );
    v1.push_back( tanh ( vc5 ) );
    complex <double> vc6 ( polar ( 1.0, pi ) );
    v1.push_back( tanh ( vc6 ) );

    cout << "The complex components tanh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << endl;
}
```

```
Complex number c1 = (3,4)
Complex number c2 = tanh ( c1 ) = (1.00071,0.00490826)
The modulus of c2 is: 1.00072
The argument of c2 is: 0.00490474 radians, which is 0.281021 degrees.
```

```
The complex components tanh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.792403,0.24356)
(0.85004,0.713931)
(-3.54238e-013,1.55741)
(-0.85004,0.713931)
(-0.792403,0.24356)
(-0.761594,-8.68604e-014)
```

See also

[<complex>](#)

<complex> operators

11/9/2018 • 24 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator>></code>	<code>operator<<</code>
<code>operator*</code>	<code>operator+</code>	<code>operator-</code>
<code>operator/</code>	<code>operator==</code>	

operator!=

Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template <class Type>
bool operator!=(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
bool operator!=(
    const complex<Type>& left,
    const Type& right);

template <class Type>
bool operator!=(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left

A complex number or object of its parameter type to be tested for inequality.

right

A complex number or object of its parameter type to be tested for inequality.

Return Value

true if the numbers are not equal; **false** if numbers are equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_NE.cpp
// compile with: /EHsc
#include <complex>
```



```

#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> compared with type complex<double>
    complex <double> c11 ( polar (3.0, pi / 6 ) );
    complex <double> cr1a ( polar (3.0, pi /6 ) );
    complex <double> cr1b ( polar (2.0, pi / 3 ) );

    cout << "The left-side complex number is c11 = " << c11 << endl;
    cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
    cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
    if ( c11 != cr1a )
        cout << "The complex numbers c11 & cr1a are not equal." << endl;
    else
        cout << "The complex numbers c11 & cr1a are equal." << endl;
    if ( c11 != cr1b )
        cout << "The complex numbers c11 & cr1b are not equal." << endl;
    else
        cout << "The complex numbers c11 & cr1b are equal." << endl;
    cout << endl;

    // Example of the second member function
    // type complex<int> compared with type int
    complex <int> c12a ( 3, 4 );
    complex <int> c12b ( 5,0 );
    int cr2a =3;
    int cr2b =5;

    cout << "The 1st left-side complex number is c12a = " << c12a << endl;
    cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
    if ( c12a != cr2a )
        cout << "The complex numbers c12a & cr2a are not equal." << endl;
    else
        cout << "The complex numbers c12a & cr2a are equal." << endl;

    cout << "The 2nd left-side complex number is c12b = " << c12b << endl;
    cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
    if ( c12b != cr2b )
        cout << "The complex numbers c12b & cr2b are not equal." << endl;
    else
        cout << "The complex numbers c12b & cr2b are equal." << endl;
    cout << endl;

    // Example of the third member function
    // type double compared with type complex<double>
    double c13a =3;
    double c13b =5;
    complex <double> cr3a ( 3, 4 );
    complex <double> cr3b ( 5,0 );

    cout << "The 1st left-side complex number is c13a = " << c13a << endl;
    cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
    if ( c13a != cr3a )
        cout << "The complex numbers c13a & cr3a are not equal." << endl;
    else
        cout << "The complex numbers c13a & cr3a are equal." << endl;

    cout << "The 2nd left-side complex number is c13b = " << c13b << endl;
    cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
    if ( c13b != cr3b )
        cout << "The complex numbers c13b & cr3b are not equal." << endl;
    else
        cout << "The complex numbers c13b & cr3b are equal." << endl;
    cout << endl;
}

```

```
}
```

```
The left-side complex number is c11 = (2.59808,1.5)
The 1st right-side complex number is cr1a = (2.59808,1.5)
The 2nd right-side complex number is cr1b = (1,1.73205)
The complex numbers c11 & cr1a are equal.
The complex numbers c11 & cr1b are not equal.
```

```
The 1st left-side complex number is c12a = (3,4)
The 1st right-side complex number is cr2a = 3
The complex numbers c12a & cr2a are not equal.
The 2nd left-side complex number is c12b = (5,0)
The 2nd right-side complex number is cr2b = 5
The complex numbers c12b & cr2b are equal.
```

```
The 1st left-side complex number is c13a = 3
The 1st right-side complex number is cr3a = (3,4)
The complex numbers c13a & cr3a are not equal.
The 2nd left-side complex number is c13b = 5
The 2nd right-side complex number is cr3b = (5,0)
The complex numbers c13b & cr3b are equal.
```

operator*

Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator*(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

Return Value

The complex number that results from the multiplication of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_mult.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> times type complex<double>
    complex <double> c11 ( polar (3.0, pi / 6 ) );
    complex <double> cr1 ( polar (2.0, pi / 3 ) );
    complex <double> cs1 = c11 * cr1;

    cout << "The left-side complex number is c11 = " << c11 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;
    cout << "Product of two complex numbers is: cs1 = " << cs1 << endl;
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> times type double
    complex <double> c12 ( polar ( 3.0, pi / 6 ) );
    double cr2 =5;
    complex <double> cs2 = c12 * cr2;

    cout << "The left-side complex number is c12 = " << c12 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "Product of two complex numbers is: cs2 = " << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double times type complex<double>
    double c13 = 5;
    complex <double> cr3 ( polar (3.0, pi / 6 ) );
    complex <double> cs3 = c13 * cr3;

    cout << "The left-side complex number is c13 = " << c13 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "Product of two complex numbers is: cs3 = " << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;
}
```

operator+

Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```

template <class Type>
complex<Type> operator+(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator+(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator+(
    const Type& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator+(const complex<Type>& left);

```

Parameters

left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

Return Value

The complex number that results from the addition of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format. The unary operator returns *left*.

Example

```

// complex_op_add.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> plus type complex<double>
    complex <double> c1 ( 3.0, 4.0 );
    complex <double> cr1 ( 2.0, 5.0 );
    complex <double> cs1 = c1 + cr1;

    cout << "The left-side complex number is c1 = " << c1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;
    cout << "The sum of the two complex numbers is: cs1 = " << cs1 << endl;
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: " << argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;
}

```

```

// Example of the second member function
// type complex<double> plus type double
complex <double> c12 ( 3.0, 4.0 );
double cr2 =5.0;
complex <double> cs2 = c12 + cr2;

cout << "The left-side complex number is c12 = " << c12 << endl;
cout << "The right-side complex number is cr2 = " << cr2 << endl;
cout << "The sum of the two complex numbers is: cs2 = " << cs2 << endl;
double abscs2 = abs ( cs2 );
double argcs2 = arg ( cs2 );
cout << "The modulus of cs2 is: " << abscs2 << endl;
cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
    << argcs2 * 180 / pi << " degrees." << endl << endl;

// Example of the third member function
// type double plus type complex<double>
double c13 = 5.0;
complex <double> cr3 ( 3.0, 4.0 );
complex <double> cs3 = c13 + cr3;

cout << "The left-side complex number is c13 = " << c13 << endl;
cout << "The right-side complex number is cr3 = " << cr3 << endl;
cout << "The sum of the two complex numbers is: cs3 = " << cs3 << endl;
double abscs3 = abs ( cs3 );
double argcs3 = arg ( cs3 );
cout << "The modulus of cs3 is: " << abscs3 << endl;
cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
    << argcs3 * 180 / pi << " degrees." << endl << endl;

// Example of the fourth member function
// plus type complex<double>
complex <double> cr4 ( 3.0, 4.0 );
complex <double> cs4 = + cr4;

cout << "The right-side complex number is cr4 = " << cr4 << endl;
cout << "The result of the unary application of + to the right-side"
    << "\n complex number is: cs4 = " << cs4 << endl;
double abscs4 = abs ( cs4 );
double argcs4 = arg ( cs4 );
cout << "The modulus of cs4 is: " << abscs4 << endl;
cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
    << argcs4 * 180 / pi << " degrees." << endl << endl;
}

```

```

The left-side complex number is c11 = (3,4)
The right-side complex number is cr1 = (2,5)
The sum of the two complex numbers is: cs1 = (5,9)
The modulus of cs1 is: 10.2956
The argument of cs1 is: 1.0637 radians, which is 60.9454 degrees.

The left-side complex number is c12 = (3,4)
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = (8,4)
The modulus of cs2 is: 8.94427
The argument of cs2 is: 0.463648 radians, which is 26.5651 degrees.

The left-side complex number is c13 = 5
The right-side complex number is cr3 = (3,4)
The sum of the two complex numbers is: cs3 = (8,4)
The modulus of cs3 is: 8.94427
The argument of cs3 is: 0.463648 radians, which is 26.5651 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of + to the right-side
complex number is: cs4 = (3,4)
The modulus of cs4 is: 5
The argument of cs4 is: 0.927295 radians, which is 53.1301 degrees.

```

operator-

Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```

template <class Type>
complex<Type> operator-(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator-(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator-(
    const Type& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator-(const complex<Type>& left);

```

Parameters

left

The first of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

right

The second of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

Return Value

The complex number that results from the subtraction of *right* from *left*, the two numbers whose values are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

The unary operator changes the sign of a complex number and returns a value whose real part is the negative of the real part of the number input and whose imaginary part is the negative of the imaginary part of the number input.

Example

```
// complex_op_sub.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> minus type complex<double>
    complex <double> c1 ( 3.0, 4.0 );
    complex <double> cr1 ( 2.0, 5.0 );
    complex <double> cs1 = c1 - cr1;

    cout << "The left-side complex number is c1 = " << c1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;
    cout << "Difference of two complex numbers is: cs1 = " << cs1 << endl;
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> minus type double
    complex <double> c2 ( 3.0, 4.0 );
    double cr2 =5.0;
    complex <double> cs2 = c2 - cr2;

    cout << "The left-side complex number is c2 = " << c2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "Difference of two complex numbers is: cs2 = " << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double minus type complex<double>
    double c3 = 5.0;
    complex <double> cr3 ( 3.0, 4.0 );
    complex <double> cs3 = c3 - cr3;

    cout << "The left-side complex number is c3 = " << c3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "Difference of two complex numbers is: cs3 = " << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;

    // Example of the fourth member function
    // minus type complex<double>
```

```
// minus type complex<double>
complex <double> cr4 ( 3.0, 4.0 );
complex <double> cs4 = - cr4;

cout << "The right-side complex number is cr4 = " << cr4 << endl;
cout << "The result of the unary application of - to the right-side"
    << "\n complex number is: cs4 = " << cs4 << endl;
double abscs4 = abs ( cs4 );
double argcs4 = arg ( cs4 );
cout << "The modulus of cs4 is: " << abscs4 << endl;
cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
    << argcs4 * 180 / pi << " degrees." << endl << endl;
}
```

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,5)
Difference of two complex numbers is: cs1 = (1,-1)
The modulus of cs1 is: 1.41421
The argument of cs1 is: -0.785398 radians, which is -45 degrees.

The left-side complex number is cl2 = (3,4)
The right-side complex number is cr2 = 5
Difference of two complex numbers is: cs2 = (-2,4)
The modulus of cs2 is: 4.47214
The argument of cs2 is: 2.03444 radians, which is 116.565 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (3,4)
Difference of two complex numbers is: cs3 = (2,-4)
The modulus of cs3 is: 4.47214
The argument of cs3 is: -1.10715 radians, which is -63.4349 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of - to the right-side
complex number is: cs4 = (-3,-4)
The modulus of cs4 is: 5
The argument of cs4 is: -2.2143 radians, which is -126.87 degrees.
```

operator/

Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator*(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left

A complex number or a number that is of the parameter type for a complex number that is the numerator to be divided by the denominator with the / operation.

right

A complex number or a number that is of the parameter type for a complex number that is the denominator to be used to divide the numerator with the / operation.

Return Value

The complex number that results from the division of the numerator by the denominator, the values of which are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```

// complex_op_div.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> divided by type complex<double>
    complex <double> c1 ( polar ( 3.0, pi / 6 ) );
    complex <double> cr1 ( polar ( 2.0, pi / 3 ) );
    complex <double> cs1 = c1 / cr1;

    cout << "The left-side complex number is c1 = " << c1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;
    cout << "The quotient of the two complex numbers is: cs1 = c1 / cr1 = "
        << cs1 << endl;
    double abscs1 = abs ( cs1 );
    double argcs1 = arg ( cs1 );
    cout << "The modulus of cs1 is: " << abscs1 << endl;
    cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

    // example of the second member function
    // type complex<double> divided by type double
    complex <double> c2 ( polar (3.0, pi / 6 ) );
    double cr2 =5;
    complex <double> cs2 = c2 / cr2;

    cout << "The left-side complex number is c2 = " << c2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;
    cout << "The quotient of the two complex numbers is: cs2 = c2 / cr2 = "
        << cs2 << endl;
    double abscs2 = abs ( cs2 );
    double argcs2 = arg ( cs2 );
    cout << "The modulus of cs2 is: " << abscs2 << endl;
    cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

    // Example of the third member function
    // type double divided by type complex<double>
    double c3 = 5;
    complex <double> cr3 ( polar ( 3.0, pi / 6 ) );
    complex <double> cs3 = c3 / cr3;

    cout << "The left-side complex number is c3 = " << c3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "The quotient of the two complex numbers is: cs3 = c3 / cr2 = "
        << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;
}

```

The left-side complex number is $cl1 = (2.59808, 1.5)$
The right-side complex number is $cr1 = (1, 1.73205)$
The quotient of the two complex numbers is: $cs1 = cl1 / cr1 = (1.29904, -0.75)$
The modulus of $cs1$ is: 1.5
The argument of $cs1$ is: -0.523599 radians, which is -30 degrees.

The left-side complex number is $cl2 = (2.59808, 1.5)$
The right-side complex number is $cr2 = 5$
The quotient of the two complex numbers is: $cs2 = cl2 / cr2 = (0.519615, 0.3)$
The modulus of $cs2$ is: 0.6
The argument of $cs2$ is: 0.523599 radians, which is 30 degrees.

The left-side complex number is $cl3 = 5$
The right-side complex number is $cr3 = (2.59808, 1.5)$
The quotient of the two complex numbers is: $cs3 = cl3 / cr3 = (1.44338, -0.833333)$
The modulus of $cs3$ is: 1.66667
The argument of $cs3$ is: -0.523599 radians, which is -30 degrees.

operator<<

Inserts a complex number specified into the output stream.

```
template <class Type, class Elem, class Traits>
basic_ostream<Elem, Traits>& operator<<(  
    basic_ostream<Elem, Traits>& Ostr,  
    const complex<Type>& right);
```

Parameters

Ostr

The output stream into which the complex number is being entered.

right

The complex number to be entered into the output stream

Return Value

Writes the value of the specified complex number to the *Ostr* in a Cartesian format: (*real part*, *imaginary part*).

Remarks

The output stream is overloaded so that it will accept any form of a complex number, and its default output format is the Cartesian format.

Example

```
// complex_op_insert.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    complex <double> c1 ( 3.0, 4.0 );
    cout << "Complex number c1 = " << c1 << endl;

    complex <double> c2 ( polar ( 2.0, pi / 6 ) );
    cout << "Complex number c2 = " << c2 << endl;

    // To display in polar form
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is: " << absc2 << endl;
    cout << "The argument of c2 is: " << argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;
}
```

```
Complex number c1 = (3,4)
Complex number c2 = (1.73205,1)
The modulus of c2 is: 2
The argument of c2 is: 0.523599 radians, which is 30 degrees.
```

operator==

Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
template <class Type>
bool operator==(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
bool operator==(
    const complex<Type>& left,
    const Type& right);

template <class Type>
bool operator==(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left

A complex number or object of its parameter type to be tested for inequality.

right

A complex number or object of its parameter type to be tested for inequality.

Return Value

true if the numbers are equal; **false** if numbers are not equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_EQ.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> compared with type complex<double>
    complex <double> c11 ( polar ( 3.0, pi / 6 ) );
    complex <double> cr1a ( polar ( 3.0, pi / 6 ) );
    complex <double> cr1b ( polar ( 2.0, pi / 3 ) );

    cout << "The left-side complex number is c11 = " << c11 << endl;
    cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
    cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
    if ( c11 == cr1a )
        cout << "The complex numbers c11 & cr1a are equal." << endl;
    else
        cout << "The complex numbers c11 & cr1a are not equal." << endl;
    if ( c11 == cr1b )
        cout << "The complex numbers c11 & cr1b are equal." << endl;
    else
        cout << "The complex numbers c11 & cr1b are not equal." << endl;
    cout << endl;

    // Example of the second member function
    // type complex<int> compared with type int
    complex <int> c12a ( 3, 4 );
    complex <int> c12b ( 5, 0 );
    int cr2a = 3;
    int cr2b = 5;

    cout << "The 1st left-side complex number is c12a = " << c12a << endl;
    cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
    if ( c12a == cr2a )
        cout << "The complex numbers c12a & cr2a are equal." << endl;
    else
        cout << "The complex numbers c12a & cr2a are not equal." << endl;

    cout << "The 2nd left-side complex number is c12b = " << c12b << endl;
    cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
    if ( c12b == cr2b )
        cout << "The complex numbers c12b & cr2b are equal." << endl;
    else
        cout << "The complex numbers c12b & cr2b are not equal." << endl;
    cout << endl;

    // Example of the third member function
    // type double compared with type complex<double>
    double c13a = 3;
    double c13b = 5;
    complex <double> cr3a ( 3, 4 );
    complex <double> cr3b ( 5, 0 );
```

```

cout << "The 1st left-side complex number is c13a = " << c13a << endl;
cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
if ( c13a == cr3a )
    cout << "The complex numbers c13a & cr3a are equal." << endl;
else
    cout << "The complex numbers c13a & cr3a are not equal." << endl;

cout << "The 2nd left-side complex number is c13b = " << c13b << endl;
cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
if ( c13b == cr3b )
    cout << "The complex numbers c13b & cr3b are equal." << endl;
else
    cout << "The complex numbers c13b & cr3b are not equal." << endl;
cout << endl;
}

```

The left-side complex number is c11 = (2.59808,1.5)
 The 1st right-side complex number is cr1a = (2.59808,1.5)
 The 2nd right-side complex number is cr1b = (1,1.73205)
 The complex numbers c11 & cr1a are equal.
 The complex numbers c11 & cr1b are not equal.

The 1st left-side complex number is c12a = (3,4)
 The 1st right-side complex number is cr2a = 3
 The complex numbers c12a & cr2a are not equal.
 The 2nd left-side complex number is c12b = (5,0)
 The 2nd right-side complex number is cr2b = 5
 The complex numbers c12b & cr2b are equal.

The 1st left-side complex number is c13a = 3
 The 1st right-side complex number is cr3a = (3,4)
 The complex numbers c13a & cr3a are not equal.
 The 2nd left-side complex number is c13b = 5
 The 2nd right-side complex number is cr3b = (5,0)
 The complex numbers c13b & cr3b are equal.

operator>>

Extracts a complex value from the input stream.

```

template <class Type, class Elem, class Traits>
basic_istream<Elem, Traits>& operator>>(
    basic_istream<Elem, Traits>& Istr,
    complex<Type>& right);

```

Parameters

Istr

The input stream from which the complex number is being extracted.

right

The complex number that is being extracted from the input stream.

Return Value

Reads the value of the specified complex number from *Istr* and returns it into *right*.

Remarks

The valid input formats are

- *(real part, imaginary part)*

- *(real part)*
- *real part*

Example

```
// complex_op_extract.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    complex <double> c2;

    cout << "Input a complex number ( try: 2.0 ): ";
    cin >> c2;
    cout << c2 << endl;
}
```

```
Input a complex number ( try: 2.0 ): 2.0
2.0
```

See also

[<complex>](#)

complex Class

3/28/2019 • 18 minutes to read • [Edit Online](#)

The template class describes an object that stores two objects of type `Type`, one that represents the real part of a complex number and one that represents the imaginary part.

Syntax

```
template <class Type>
class complex
```

Remarks

An object of class `Type`:

- Has a public default constructor, destructor, copy constructor, and assignment operator with conventional behavior.
- Can be assigned integer or floating-point values, or type cast to such values with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class `Type` may throw exceptions.

Explicit specializations of template class `complex` exist for the three floating-point types. In this implementation, a value of any other type `Type` is typecast to **double** for actual calculations, with the **double** result assigned back to the stored object of type `Type`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>complex</code>	Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

Typedefs

TYPE NAME	DESCRIPTION
<code>value_type</code>	A type that represents the data type used to represent the real and imaginary parts of a complex number.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>imag</code>	Extracts the imaginary component of a complex number.
<code>real</code>	Extracts the real component of a complex number.

Operators

OPERATOR	DESCRIPTION
<code>operator* =</code>	Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
<code>operator+ =</code>	Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.
<code>operator- =</code>	Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.
<code>operator/=</code>	Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
<code>operator=</code>	Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it is being assigned.

Requirements

Header: <complex>

Namespace: std

complex::complex

Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

```
constexpr complex(  
    const T& _RealVal = 0,  
    const T& _ImagVal = 0);  
  
template <class Other>  
constexpr complex(  
    const complex<Other>& complexNum);
```

Parameters

_RealVal

The value of the real part used to initialize the complex number being constructed.

_ImagVal

The value of the imaginary part used to initialize the complex number being constructed.

complexNum

The complex number whose real and imaginary parts are used to initialize the complex number being constructed.

Remarks

The first constructor initializes the stored real part to *_RealVal* and the stored imaginary part to *_ImagVal*. The second constructor initializes the stored real part to `complexNum.real()` and the stored imaginary part to `complexNum.imag()`.

In this implementation, if a translator does not support member template functions, the template:

```
template <class Other>
complex(const complex<Other>& right);
```

is replaced with:

```
complex(const complex& right);
```

which is the copy constructor.

Example

```
// complex_complex.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex<double> c1( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,"
         << "c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of another complex number
    complex<double> c2( c1 );
    cout << "Initializing with the real and imaginary parts of c1,"
         << " c2 = " << c2 << endl;

    // Complex numbers can be initialized in polar form
    // but will be stored in Cartesian form
    complex<double> c3( polar( sqrt( (double)8 ) , pi / 4 ) );
    cout << "c3 = polar( sqrt( 8 ) , pi / 4 ) = " << c3 << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs( c3 );
    double argc3 = arg( c3 );
    cout << "The modulus of c3 is recovered from c3 using: abs( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

complex::imag

Extracts the imaginary component of a complex number.

```
T imag() const;

T imag(const T& right);
```

Parameters

right

A complex number whose imaginary value is to be extracted.

Return Value

The imaginary part of the complex number.

Remarks

For a complex number $a + bi$, the imaginary part or component is $Im(a + bi) = b$.

Example

```
// complex_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;

    complex<double> c1( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = c1.real();
    cout << "The real part of c1 is c1.real() = "
        << dr1 << "." << endl;

    double di1 = c1.imag();
    cout << "The imaginary part of c1 is c1.imag() = "
        << di1 << "." << endl;
}
```

```
The complex number c1 = (4,3)
The real part of c1 is c1.real() = 4.
The imaginary part of c1 is c1.imag() = 3.
```

complex::operator*=

Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

```
template <class Other>
complex& operator*=(const complex<Other>& right);

complex<Type>& operator*=(const Type& right);

complex<Type>& operator*=(const complex<Type>& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been multiplied by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
// complex_op_me.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main()
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> multiplied by type complex<double>
    complex<double> c1( polar ( 3.0 , pi / 6 ) );
    complex<double> cr1( polar ( 2.0 , pi / 3 ) );
    cout << "The left-side complex number is c1 = " << c1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = c1 * cr1;
    cout << "Quotient of two complex numbers is: cs1 = c1 * cr1 = "
         << cs1 << endl;

    // This is equivalent to the following operation
    c1 *= cr1;
    cout << "Quotient of two complex numbers is also: c1 *= cr1 = "
         << c1 << endl;

    double abscl1 = abs ( c1 );
    double argcl1 = arg ( c1 );
    cout << "The modulus of c1 is: " << abscl1 << endl;
    cout << "The argument of c1 is: "<< argcl1 << " radians, which is "
         << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> multiplied by type double
    complex<double> c2 ( polar ( 3.0 , pi / 6 ) );
    double cr2 = 5.0;
    cout << "The left-side complex number is c2 = " << c2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = c2 * cr2;
    cout << "Quotient of two complex numbers is: cs2 = c2 * cr2 = "
         << cs2 << endl;

    // This is equivalent to the following operation
    c2 *= cr2;
    cout << "Quotient of two complex numbers is also: c2 *= cr2 = "
         << c2 << endl;

    double abscl2 = abs ( c2 );
    double argcl2 = arg ( c2 );
    cout << "The modulus of c2 is: " << abscl2 << endl;
    cout << "The argument of c2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl;
}
```

complex::operator+=

Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.

```
template <class Other>
complex<Type>& operator+=(const complex<Other>& right);

complex<Type>& operator+=(const Type& right);

complex<Type>& operator+=(const complex<Type>& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has had the number specified as a parameter added.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```

// complex_op_pe.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> added to type complex<double>
    complex<double> cl1( 3.0 , 4.0 );
    complex<double> cr1( 2.0 , -1.0 );
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = cl1 + cr1;
    cout << "The sum of the two complex numbers is: cs1 = cl1 + cr1 = "
        << cs1 << endl;

    // This is equivalent to the following operation
    cl1 += cr1;
    cout << "The complex number cr1 added to the complex number cl1 is:"
        << "\n cl1 += cr1 = " << cl1 << endl;

    double abscl1 = abs( cl1 );
    double argcl1 = arg( cl1 );
    cout << "The modulus of cl1 is: " << abscl1 << endl;
    cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type double added to type complex<double>
    complex<double> cl2( -2 , 4 );
    double cr2 = 5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = cl2 + cr2;
    cout << "The sum of the two complex numbers is: cs2 = cl2 + cr2 = "
        << cs2 << endl;

    // This is equivalent to the following operation
    cl2 += cr2;
    cout << "The complex number cr2 added to the complex number cl2 is:"
        << "\n cl2 += cr2 = " << cl2 << endl;

    double abscl2 = abs( cl2 );
    double argcl2 = arg( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
        << argcl2 * 180 / pi << " degrees." << endl << endl;
}

```

```
The left-side complex number is c11 = (3,4)
The right-side complex number is cr1 = (2,-1)
The sum of the two complex numbers is: cs1 = c11 + cr1 = (5,3)
The complex number cr1 added to the complex number c11 is:
c11 += cr1 = (5,3)
The modulus of c11 is: 5.83095
The argument of c11 is: 0.54042 radians, which is 30.9638 degrees.

The left-side complex number is c12 = (-2,4)
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = c12 + cr2 = (3,4)
The complex number cr2 added to the complex number c12 is:
c12 += cr2 = (3,4)
The modulus of c12 is: 5
The argument of c12 is: 0.927295 radians, which is 53.1301 degrees.
```

complex::operator-=

Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it is added.

```
template <class Other>
complex<Type>& operator-=(const complex<Other>& complexNum);

complex<Type>& operator-=(const Type& _RealPart);

complex<Type>& operator-=(const complex<Type>& complexNum);
```

Parameters

complexNum

A complex number to be subtracted from the target complex number.

_RealPart

A real number to be subtracted from the target complex number.

Return Value

A complex number that has had the number specified as a parameter subtracted from it.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```

// complex_op_se.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> subtracted from type complex<double>
    complex<double> cl1( 3.0 , 4.0 );
    complex<double> cr1( 2.0 , -1.0 );
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = cl1 - cr1;
    cout << "The difference between the two complex numbers is:"
        << "\n cs1 = cl1 - cr1 = " << cs1 << endl;

    // This is equivalent to the following operation
    cl1 -= cr1;
    cout << "Complex number cr1 subtracted from complex number cl1 is:"
        << "\n cl1 -= cr1 = " << cl1 << endl;

    double abscl1 = abs( cl1 );
    double argcl1 = arg( cl1 );
    cout << "The modulus of cl1 is: " << abscl1 << endl;
    cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type double subtracted from type complex<double>
    complex<double> cl2( 2.0 , 4.0 );
    double cr2 = 5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = cl2 - cr2;
    cout << "The difference between the two complex numbers is:"
        << "\n cs2 = cl2 - cr2 = " << cs2 << endl;

    // This is equivalent to the following operation
    cl2 -= cr2;
    cout << "Complex number cr2 subtracted from complex number cl2 is:"
        << "\n cl2 -= cr2 = " << cl2 << endl;

    double abscl2 = abs( cl2 );
    double argcl2 = arg( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
        << argcl2 * 180 / pi << " degrees." << endl << endl;
}

```



```
The left-side complex number is c11 = (3,4)
The right-side complex number is cr1 = (2,-1)
The difference between the two complex numbers is:
cs1 = c11 - cr1 = (1,5)
Complex number cr1 subtracted from complex number c11 is:
c11 -= cr1 = (1,5)
The modulus of c11 is: 5.09902
The argument of c11 is: 1.3734 radians, which is 78.6901 degrees.
```

```
The left-side complex number is c12 = (2,4)
The right-side complex number is cr2 = 5
The difference between the two complex numbers is:
cs2 = c12 - cr2 = (-3,4)
Complex number cr2 subtracted from complex number c12 is:
c12 -= cr2 = (-3,4)
The modulus of c12 is: 5
The argument of c12 is: 2.2143 radians, which is 126.87 degrees.
```

complex::operator/=

Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

```
template <class Other>
complex<Type>& operator/=(const complex<Other>& complexNum);

complex<Type>& operator/=(const Type& _RealPart);

complex<Type>& operator/=(const complex<Type>& complexNum);
```

Parameters

complexNum

A complex number to be subtracted from the target complex number.

_RealPart

A real number to be subtracted from the target complex number.

Return Value

A complex number that has been divided by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```

// complex_op_de.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> divided by type complex<double>
    complex<double> c1( polar (3.0 , pi / 6 ) );
    complex<double> cr1( polar (2.0 , pi / 3 ) );
    cout << "The left-side complex number is c1 = " << c1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = c1 / cr1;
    cout << "The quotient of the two complex numbers is: cs1 = c1 / cr1 = "
        << cs1 << endl;

    // This is equivalent to the following operation
    c1 /= cr1;
    cout << "Quotient of two complex numbers is also: c1 /= cr1 = "
        << c1 << endl;

    double abscl1 = abs( c1 );
    double argcl1 = arg( c1 );
    cout << "The modulus of c1 is: " << abscl1 << endl;
    cout << "The argument of c1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> divided by type double
    complex<double> c2( polar(3.0 , pi / 6 ) );
    double cr2 =5;
    cout << "The left-side complex number is c2 = " << c2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = c2 / cr2;
    cout << "The quotient of the two complex numbers is: cs2 /= c2 cr2 = "
        << cs2 << endl;

    // This is equivalent to the following operation
    c2 /= cr2;
    cout << "Quotient of two complex numbers is also: c2 = /cr2 = "
        << c2 << endl;

    double abscl2 = abs( c2 );
    double argcl2 = arg( c2 );
    cout << "The modulus of c2 is: " << abscl2 << endl;
    cout << "The argument of c2 is: "<< argcl2 << " radians, which is "
        << argcl2 * 180 / pi << " degrees." << endl << endl;
}

```

```
The left-side complex number is c11 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
The quotient of the two complex numbers is: cs1 = c11 /cr1 = (1.29904,-0.75)
Quotient of two complex numbers is also: c11 /= cr1 = (1.29904,-0.75)
The modulus of c11 is: 1.5
The argument of c11 is: -0.523599 radians, which is -30 degrees.

The left-side complex number is c12 = (2.59808,1.5)
The right-side complex number is cr2 = 5
The quotient of the two complex numbers is: cs2 /= c12 cr2 = (0.519615,0.3)
Quotient of two complex numbers is also: c12 = /cr2 = (0.519615,0.3)
The modulus of c12 is: 0.6
The argument of c12 is: 0.523599 radians, which is 30 degrees.
```

complex::operator=

Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it is being assigned.

```
template <class Other>
complex<Type>& operator=(const complex<Other>& right);

complex<Type>& operator=(const Type& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been assigned the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```

// complex_op_as.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> assigned to type complex<double>
    complex<double> cl1( 3.0 , 4.0 );
    complex<double> cr1( 2.0 , -1.0 );
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    cl1 = cr1;
    cout << "The complex number cr1 assigned to the complex number cl1 is:"
        << "\ncl1 = cr1 = " << cl1 << endl;

    // Example of the second member function
    // type double assigned to type complex<double>
    complex<double> cl2( -2 , 4 );
    double cr2 =5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    cl2 = cr2;
    cout << "The complex number cr2 assigned to the complex number cl2 is:"
        << "\ncl2 = cr2 = " << cl2 << endl;

    cl2 = complex<double>(3.0, 4.0);
    cout << "The complex number (3, 4) assigned to the complex number cl2 is:"
        << "\ncl2 = " << cl2 << endl;
}

```

```

The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The complex number cr1 assigned to the complex number cl1 is:
cl1 = cr1 = (2,-1)
The left-side complex number is cl2 = (-2,4)
The right-side complex number is cr2 = 5
The complex number cr2 assigned to the complex number cl2 is:
cl2 = cr2 = (5,0)
The complex number (3, 4) assigned to the complex number cl2 is:
cl2 = (3,4)

```

complex::real

Gets or sets the real component of a complex number.

```

constexpr T real() const;

T real(const T& right);

```

Parameters

right

A complex number whose real value is to be extracted.

Return Value

The real part of the complex number.

Remarks

For a complex number $a + bi$, the real part or component is $Re(a + bi) = a$.

Example

```
// complex_class_real.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;

    complex<double> c1( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = c1.real();
    cout << "The real part of c1 is c1.real() = "
        << dr1 << "." << endl;

    double di1 = c1.imag();
    cout << "The imaginary part of c1 is c1.imag() = "
        << di1 << "." << endl;
}
```

```
The complex number c1 = (4,3)
The real part of c1 is c1.real() = 4.
The imaginary part of c1 is c1.imag() = 3.
```

complex::value_type

A type that represents the data type used to represent the real and imaginary parts of a complex number.

```
typedef Type value_type;
```

Remarks

`value_type` is a synonym for the class `complex` `Type` template parameter.

Example

```
// complex_valuetype.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    complex<double>::value_type a = 3, b = 4;

    complex<double> c1 ( a , b );
    cout << "Specifying initial real & imaginary parts"
        << "\nof type value_type: "
        << "c1 = " << c1 << "." << endl;
}
```

```
Specifying initial real & imaginary parts  
of type value_type: c1 = (3,4).
```

See also

[Thread Safety in the C++ Standard Library](#)

complex<double>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that stores an ordered pair of objects both of type **double**, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

```
template <>
class complex<double> {
public:
    constexpr complex(
        double RealVal = 0,
        double ImagVal = 0);

    constexpr complex(const complex<double>& complexNum);

    constexpr explicit complex(const complex<long double>& complexNum);
    // rest same as template class complex
};
```

Parameters

RealVal

The value of type **double** for the real part of the complex number being constructed.

ImagVal

The value of type **double** for the imaginary part of the complex number being constructed.

complexNum

The complex number of type **float** or of type **long double** whose real and imaginary parts are used to initialize a complex number of type **double** being constructed.

Return Value

A complex number of type **double**.

Remarks

The explicit specialization of the template class `complex` to a complex class of type **double** differs from the template class only in the constructors it defines. The conversion from **float** to **double** is allowed to be implicit, but the conversion from **long double** to **double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class `complex`, see [complex Class](#). For a list of members of the template class `complex`, see .

Example

```

// complex_comp_dbl.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex <double> c1 ( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,\n"
         << "as type double gives c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type float
    complex <float> c2float ( 4.0 , 5.0 );
    complex <double> c2double ( c2float );
    cout << "Implicit conversion from type float to type double,"
         << endl << "gives c2double = " << c2double << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type long double
    complex <long double> c3longdouble ( 4.0 , 5.0 );
    complex <double> c3double ( c3longdouble );
    cout << "Explicit conversion from type float to type double,"
         << endl << "gives c3longdouble = " << c3longdouble << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3longdouble );
    double argc3 = arg ( c3longdouble );
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:" << endl
         << "arg ( c3 ) = " << argc3 << " radians, which is "
         << argc3 * 180 / pi << " degrees." << endl;
}
/* Output:
Specifying initial real & imaginary parts,
as type double gives c1 = (4,5)
Implicit conversion from type float to type double,
gives c2double = (4,5)
Explicit conversion from type float to type double,
gives c3longdouble = (4,5)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 6.40312
Argument of c3 is recovered from c3 using:
arg ( c3 ) = 0.896055 radians, which is 51.3402 degrees.
*/

```

Requirements

Header: <complex>

Namespace: std

See also

[complex Class](#)

[Thread Safety in the C++ Standard Library](#)

complex<float>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that stores an ordered pair of objects both of type **float**, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

```
template <>
class complex<float> {
public:
    constexpr complex(
        float _RealVal = 0,
        float _ImagVal = 0);

    constexpr complex(
        const complex<float>& complexNum);

    constexpr complex(
        const complex<double>& complexNum);

    constexpr complex(
        const complex<long double>& complexNum);
    // rest same as template class complex
};
```

Parameters

_RealVal

The value of type **float** for the real part of the complex number being constructed.

_ImagVal

The value of type **float** for the imaginary part of the complex number being constructed.

complexNum

The complex number of type **double** or of type **long double** whose real and imaginary parts are used to initialize a complex number of type **float** being constructed.

Return Value

A complex number of type **float**.

Remarks

The explicit specialization of the template class `complex` to a complex class of type **float** differs from the template class only in the constructors it defines. The conversion from **float** to **double** is allowed to be implicit, but the less safe conversion from **float** to **long double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class `complex`, see [complex Class](#). For a list of members of the template class `complex`, see .

Example

```

// complex_comp_flt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex <float> c1 ( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type double
    complex <double> c2double ( 1.0 , 3.0 );
    complex <float> c2float ( c2double );
    cout << "Implicit conversion from type double to type float,"
        << endl << "gives c2float = " << c2float << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type long double
    complex <long double> c3longdouble ( 3.0 , 4.0 );
    complex <float> c3float ( c3longdouble );
    cout << "Explicit conversion from type long double to type float,"
        << endl << "gives c3float = " << c3float << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs ( c3float);
    double argc3 = arg ( c3float);
    cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
        << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:"
        << endl << "arg ( c3 ) = "
        << argc3 << " radians, which is " << argc3 * 180 / pi
        << " degrees." << endl;
}
/* Output:
Specifying initial real & imaginary parts,
as type float gives c1 = (4,5)
Implicit conversion from type double to type float,
gives c2float = (1,3)
Explicit conversion from type long double to type float,
gives c3float = (3,4)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 5
Argument of c3 is recovered from c3 using:
arg ( c3 ) = 0.927295 radians, which is 53.1301 degrees.
*/

```

Requirements

Header: <complex>

Namespace: std

See also

[complex Class](#)

[Thread Safety in the C++ Standard Library](#)

complex<long double>

10/31/2018 • 2 minutes to read • [Edit Online](#)

This explicitly specialized template class describes an object that stores an ordered pair of objects, both of type **long double**, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

```
template <>
class complex<long double> {
public:
    constexpr complex(
        long double _RealVal = 0,
        long double _ImagVal = 0);

    complex(
        constexpr complex<long double>& complexNum);

    // rest same as template class complex
};
```

Parameters

_RealVal

The value of type **long double** for the real part of the complex number being constructed.

_ImagVal

The value of type **long double** for the imaginary part of the complex number being constructed.

complexNum

The complex number of type **double** or of type **float** whose real and imaginary parts are used to initialize a complex number of type **long double** being constructed.

Return Value

A complex number of type **long double**.

Remarks

The explicit specialization of the template class `complex` to a complex class of type **long double** differs from the template class only in the constructors it defines. The conversion from **long double** to **float** is allowed to be implicit, but the conversion from **double** to **long double** is required to be **explicit**. The use of **explicit** rules out the initiation with type conversion using assignment syntax.

For more information on the template class `complex` and its members, see [complex Class](#).

Microsoft specific: The **long double** and **double** types have the same representation, but are distinct types. For more information, see [Fundamental types](#).

Example

```

// complex_comp_ld.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex<long double> c1( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type float
    complex<float> c2float( 1.0 , 3.0 );
    complex<long double> c2longdouble ( c2float );
    cout << "Implicit conversion from type float to type long double,"
        << "\n gives c2longdouble = " << c2longdouble << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type double
    complex<double> c3double( 3.0 , 4.0 );
    complex<long double> c3longdouble( c3double );
    cout << "Implicit conversion from type long double to type float,"
        << "\n gives c3longdouble = " << c3longdouble << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs( c3longdouble );
    double argc3 = arg( c3longdouble );
    cout << "The modulus of c3 is recovered from c3 using: abs( c3 ) = "
        << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg( c3 ) = "
        << argc3 << " radians, which is " << argc3 * 180 / pi
        << " degrees." << endl;
}

```

```

Specifying initial real & imaginary parts,
as type float gives c1 = (4,5)
Implicit conversion from type float to type long double,
gives c2longdouble = (1,3)
Implicit conversion from type long double to type float,
gives c3longdouble = (3,4)
The modulus of c3 is recovered from c3 using: abs( c3 ) = 5
Argument of c3 is recovered from c3 using:
arg( c3 ) = 0.927295 radians, which is 53.1301 degrees.

```

Requirements

Header: <complex>

Namespace: std

See also

[complex Class](#)

[Thread Safety in the C++ Standard Library](#)

<condition_variable>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the classes `condition_variable` and `condition_variable_any` that are used to create objects that wait for a condition to become true.

This header uses Concurrency Runtime (ConcRT) so that you can use it together with other ConcRT mechanisms. For more information about ConcRT, see [Concurrency Runtime](#).

Syntax

```
#include <condition_variable>
```

NOTE

In code that is compiled by using `/clr`, this header is blocked.

Remarks

Code that waits for a condition variable must also use a `mutex`. A calling thread must lock the `mutex` before it calls the functions that wait for the condition variable. The `mutex` is then locked when the called function returns. The `mutex` is not locked while the thread waits for the condition to become true. So that there are no unpredictable results, each thread that waits for a condition variable must use the same `mutex` object.

Objects of type `condition_variable_any` can be used with a mutex of any type. The type of the mutex that is used does not have to provide the `try_lock` method. Objects of type `condition_variable` can only be used with a mutex of type `unique_lock<mutex>`. Objects of this type may be faster than objects of type `condition_variable_any<unique_lock<mutex>>`.

To wait for an event, first lock the mutex, and then call one of the `wait` methods on the condition variable. The `wait` call blocks until another thread signals the condition variable.

Spurious wakeups occur when threads that are waiting for condition variables become unblocked without appropriate notifications. To recognize such spurious wakeups, code that waits for a condition to become true should explicitly check that condition when the code returns from a wait function. This is usually done by using a loop; you can use `wait(unique_lock<mutex>& lock, Predicate pred)` to perform this loop for you.

```
while (condition is false)
    wait for condition variable;
```

The `condition_variable_any` and `condition_variable` classes each have three methods that wait for a condition.

- `wait` waits for an unbounded time period.
- `wait_until` waits until a specified `time`.
- `wait_for` waits for a specified `time interval`.

Each of these methods has two overloaded versions. One just waits and can wake up spuriously. The other takes an additional template argument that defines a predicate. The method does not return until the predicate is **true**.

Each class also has two methods that are used to notify a condition variable that its condition is **true**.

- `notify_one` wakes up one of the threads that is waiting for the condition variable.
- `notify_all` wakes up all of the threads that are waiting for the condition variable.

See also

[Header Files Reference](#)

[condition_variable Class](#)

[condition_variable_any Class](#)

<condition_variable> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

[cv_status](#)

cv_status Enumeration

Supplies symbolic names for the return values of the methods of template class [condition_variable](#).

```
class cv_status { no_timeout timeout };
```

See also

[<condition_variable>](#)

condition_variable Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Use the `condition_variable` class to wait for an event when you have a `mutex` of type `unique_lock<mutex>`. Objects of this type may have better performance than objects of type `condition_variable_any<unique_lock<mutex>>`.

Syntax

```
class condition_variable;
```

Members

Public Constructors

NAME	DESCRIPTION
<code>condition_variable</code>	Constructs a <code>condition_variable</code> object.

Public Methods

NAME	DESCRIPTION
<code>native_handle</code>	Returns the implementation-specific type representing the <code>condition_variable</code> handle.
<code>notify_all</code>	Unblocks all threads that are waiting for the <code>condition_variable</code> object.
<code>notify_one</code>	Unblocks one of the threads that are waiting for the <code>condition_variable</code> object.
<code>wait</code>	Blocks a thread.
<code>wait_for</code>	Blocks a thread, and sets a time interval after which the thread unblocks.
<code>wait_until</code>	Blocks a thread, and sets a maximum point in time at which the thread unblocks.

Requirements

Header: `<condition_variable>`

Namespace: `std`

`condition_variable::condition_variable` Constructor

Constructs a `condition_variable` object.


```
condition_variable();
```

Remarks

If not enough memory is available, the constructor throws a [system_error](#) object that has a `not_enough_memory` error code. If the object cannot be constructed because some other resource is not available, the constructor throws a `system_error` object that has a `resource_unavailable_try_again` error code.

condition_variable::native_handle

Returns the implementation-specific type that represents the condition_variable handle.

```
native_handle_type native_handle();
```

Return Value

`native_handle_type` is defined as a pointer to C++ Concurrency Runtime internal data structures.

condition_variable::notify_all

Unblocks all threads that are waiting for the `condition_variable` object.

```
void notify_all() noexcept;
```

condition_variable::notify_one

Unblocks one of the threads that are waiting on the `condition_variable` object.

```
void notify_one() noexcept;
```

condition_variable::wait

Blocks a thread.

```
void wait(unique_lock<mutex>& Lck);

template <class Predicate>
void wait(unique_lock<mutex>& Lck, Predicate Pred);
```

Parameters

Lck

A [unique_lock<mutex>](#) object.

Pred

Any expression that returns **true** or **false**.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to [notify_one](#) or [notify_all](#). It can also wake up spuriously.

In effect, the second method executes the following code.

```
while(!Pred())
    wait(Lck);
```

condition_variable::wait_for

Blocks a thread, and sets a time interval after which the thread unblocks.

```
template <class Rep, class Period>
cv_status wait_for(
    unique_lock<mutex>& Lck,
    const chrono::duration<Rep, Period>& Rel_time);

template <class Rep, class Period, class Predicate>
bool wait_for(
    unique_lock<mutex>& Lck,
    const chrono::duration<Rep, Period>& Rel_time,
    Predicate Pred);
```

Parameters

Lck

A `unique_lock<mutex>` object.

Rel_time

A `chrono::duration` object that specifies the amount of time before the thread wakes up.

Pred

Any expression that returns **true** or **false**.

Return Value

The first method returns `cv_status::timeout` if the wait terminates when *Rel_time* has elapsed. Otherwise, the method returns `cv_status::no_timeout`.

The second method returns the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to `notify_one` or `notify_all` or until the time interval *Rel_time* has elapsed. It can also wake up spuriously.

In effect, the second method executes the following code.

```
while(!Pred())
    if(wait_for(Lck, Rel_time) == cv_status::timeout)
        return Pred();

return true;
```

condition_variable::wait_until

Blocks a thread, and sets a maximum point in time at which the thread unblocks.

```

template <class Clock, class Duration>
cv_status wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time);

template <class Clock, class Duration, class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time,
    Predicate Pred);

cv_status wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time);

template <class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time,
    Predicate Pred);

```

Parameters

Lck

A [unique_lock<mutex>](#) object.

Abs_time

A [chrono::time_point](#) object.

Pred

Any expression that returns **true** or **false**.

Return Value

Methods that return a `cv_status` type return `cv_status::timeout` if the wait terminates when *Abs_time* elapses. Otherwise, the methods return `cv_status::no_timeout`.

Methods that return a **bool** return the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to [notify_one](#) or [notify_all](#) or until `Abs_time`. It can also wake up spuriously.

In effect, the second method executes the following code

```

while(!Pred())
    if(wait_until(Lck, Abs_time) == cv_status::timeout)
        return Pred();

return true;

```

The third and fourth methods use a pointer to an object of type `xtime` to replace the `chrono::time_point` object. The `xtime` object specifies the maximum amount of time to wait for a signal.

See also

[Header Files Reference](#)

[<condition_variable>](#)

condition_variable_any Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Use the class `condition_variable_any` to wait for an event that has any `mutex` type.

Syntax

```
class condition_variable_any;
```

Members

Public Constructors

NAME	DESCRIPTION
condition_variable_any	Constructs a <code>condition_variable_any</code> object.

Public Methods

NAME	DESCRIPTION
notify_all	Unblocks all threads that are waiting for the <code>condition_variable_any</code> object.
notify_one	Unblocks one of the threads that are waiting for the <code>condition_variable_any</code> object.
wait	Blocks a thread.
wait_for	Blocks a thread, and sets a time interval after which the thread unblocks.
wait_until	Blocks a thread, and sets a maximum point in time at which the thread unblocks.

Requirements

Header: <condition_variable>

Namespace: std

condition_variable_any::condition_variable_any Constructor

Constructs a `condition_variable_any` object.

```
condition_variable_any();
```

Remarks

If not enough memory is available, the constructor throws a `system_error` object that has a `not_enough_memory` error code. If the object cannot be constructed because some other resource is not available, the constructor throws a `system_error` object that has a `resource_unavailable_try_again` error code.

`condition_variable_any::notify_all`

Unblocks all threads that are waiting for the `condition_variable_any` object.

```
void notify_all() noexcept;
```

`condition_variable_any::notify_one`

Unblocks one of the threads that are waiting on the `condition_variable_any` object.

```
void notify_one() noexcept;
```

`condition_variable_any::wait`

Blocks a thread.

```
template <class Lock>
void wait(Lock& Lck);

template <class Lock, class Predicate>
void wait(Lock& Lck, Predicate Pred);
```

Parameters

Lck

A `mutex` object of any type.

Pred

Any expression that returns **true** or **false**.

Remarks

The first method blocks until the `condition_variable_any` object is signaled by a call to `notify_one` or `notify_all`. It can also wake up spuriously.

The second method in effect executes the following code.

```
while (!Pred())
    wait(Lck);
```

`condition_variable_any::wait_for`

Blocks a thread, and sets a time interval after which the thread unblocks.

```
template <class Lock, class Rep, class Period>
bool wait_for(Lock& Lck, const chrono::duration<Rep, Period>& Rel_time);

template <class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& Lck, const chrono::duration<Rep, Period>& Rel_time, Predicate Pred);
```

Parameters

Lck

A `mutex` object of any type.

Rel_time

A `chrono::duration` object that specifies the amount of time before the thread wakes up.

Pred

Any expression that returns **true** or **false**.

Return Value

The first method returns `cv_status::timeout` if the wait terminates when *Rel_time* has elapsed. Otherwise, the method returns `cv_status::no_timeout`.

The second method returns the value of *Pred*.

Remarks

The first method blocks until the `condition_variable_any` object is signaled by a call to [notify_one](#) or [notify_all](#), or until the time interval *Rel_time* has elapsed. It can also wake up spuriously.

The second method in effect executes the following code.

```
while(!Pred())
    if(wait_for(Lck, Rel_time) == cv_status::timeout)
        return Pred();

return true;
```

condition_variable_any::wait_until

Blocks a thread, and sets a maximum point in time at which the thread unblocks.

```
template <class Lock, class Clock, class Duration>
void wait_until(Lock& Lck, const chrono::time_point<Clock, Duration>& Abs_time);

template <class Lock, class Clock, class Duration, class Predicate>
void wait_until(
    Lock& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time,
    Predicate Pred);

template <class Lock>
void wait_until(Lock Lck, const xtime* Abs_time);

template <class Lock, class Predicate>
void wait_until(
    Lock Lck,
    const xtime* Abs_time,
    Predicate Pred);
```

Parameters

Lck

A mutex object.

Abs_time

A `chrono::time_point` object.

Pred

Any expression that returns **true** or **false**.

Return Value

Methods that return a `cv_status` type return `cv_status::timeout` if the wait terminates when *Abs_time* elapses. Otherwise, the methods return `cv_status::no_timeout`.

Methods that return a `bool` return the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to [notify_one](#) or [notify_all](#), or until *Abs_time*. It can also wake up spuriously.

The second method in effect executes the following code.

```
while(!Pred())
    if(wait_until(Lck, Abs_time) == cv_status::timeout)
        return Pred();

return true;
```

The third and fourth methods use a pointer to an object of type `xtime` to replace the `chrono::time_point` object. The `xtime` object specifies the maximum amount of time to wait for a signal.

See also

[Header Files Reference](#)

[<condition_variable>](#)

<csetjmp>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <setjmp.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <csetjmp>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<csignal>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <signal.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <csignal>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstdarg>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <stdarg.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstdarg>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstdlibbool>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <stdlib.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstdlibbool>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstddef>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <stddef.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstddef>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstdlib>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header `<stdlib.h>` and adds the associated names to the `std` namespace.

Syntax

```
#include <cstdlib>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

<cstdio>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <stdio.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstdio>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstdlib>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <stdlib.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstdlib>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cstring>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <string.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cstring>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<ctgmath>

11/9/2018 • 2 minutes to read • [Edit Online](#)

In effect, includes the C++ Standard Library headers <complex> and <cmath>, which provide type-generic math macros equivalent to <tgmath.h>.

Syntax

```
#include <ctgmath>
```

Remarks

The functionality of the Standard C library header <tgmath.h> is provided by overloads in <complex> and <cmath>.

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[<complex>](#)

[<cmath>](#)

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<ctime>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <time.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <ctime>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cv/wbuffer>

11/9/2018 • 2 minutes to read • [Edit Online](#)

The header `<cv/wstring>` in previous versions of Visual Studio defined the template class [wbuffer_convert Class](#) in the `stdext::cv` namespace. The header is maintained for backward compatibility. New code should use the version of the class that is defined in [<locale>](#) in the `std` namespace

Syntax

```
#include <cv/wbuffer>
```

See also

[Header Files Reference](#)

<cv/wstring>

11/9/2018 • 2 minutes to read • [Edit Online](#)

The header `<cv/wstring>` in previous versions of Visual Studio defined the template class [wstring_convert Class](#) in the `stdext::cv` namespace. The header is maintained for backward compatibility. New code should use the version of the class that is defined in [<locale>](#) in the `std` namespace

Syntax

```
#include <cv/wstring>
```

Requirements

See also

[Header Files Reference](#)

<wchar>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <wchar.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <wchar>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<cwctype>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Includes the Standard C library header <wctype.h> and adds the associated names to the `std` namespace.

Syntax

```
#include <cwctype>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

[Header Files Reference](#)

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

<deque>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class deque and several supporting templates.

Syntax

```
#include <deque>
```

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.
<code>operator<</code>	Tests if the deque object on the left side of the operator is less than the deque object on the right side.
<code>operator<=</code>	Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.
<code>operator==</code>	Tests if the deque object on the left side of the operator is equal to the deque object on the right side.
<code>operator></code>	Tests if the deque object on the left side of the operator is greater than the deque object on the right side.
<code>operator>=</code>	Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

Functions

FUNCTION	DESCRIPTION
<code>swap</code>	Exchanges the elements of two deques.

Classes

CLASS	DESCRIPTION
<code>deque Class</code>	A template class of sequence containers that arrange elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<deque> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

swap

Exchanges the elements of two deques.

```
void swap(  
    deque<Type, Allocator>& left,  
    deque<Type, Allocator>& right,);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Example

See the example for [deque::swap](#).

See also

[<deque>](#)

<deque> operators

11/9/2018 • 5 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.

```
bool operator!=(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Return Value

true if the deque objects are not equal; **false** if the deque objects are equal.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deque objects are equal if they have the same number of elements and their respective elements have the same values.

Otherwise, they are unequal.

Example

```
// deque_op_ne.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1, c2;

    c1.push_back( 1 );
    c2.push_back( 2 );

    if ( c1 != c2 )
        cout << "The deques are not equal." << endl;
    else
        cout << "The deques are equal." << endl;
}
/* Output:
The deques are not equal.
*/
```

operator<

Tests if the deque object on the left side of the operator is less than the deque object on the right side.

```
bool operator<(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Return Value

true if the deque on the left side of the operator is less than and not equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_lt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1, c2;

    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 4 );

    c2.push_back( 1 );
    c2.push_back( 3 );

    if ( c1 < c2 )
        cout << "Deque c1 is less than deque c2." << endl;
    else
        cout << "Deque c1 is not less than deque c2." << endl;
}
/* Output:
Deque c1 is less than deque c2.
*/
```

operator<=

Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.

```
bool operator<=(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque` .

right

An object of type `deque` .

Return Value

true if the deque on the left side of the operator is less than or equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// deque_op_le.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1, c2;

    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 4 );

    c2.push_back( 1 );
    c2.push_back( 3 );

    if ( c1 <= c2 )
        cout << "Deque c1 is less than or equal to deque c2." << endl;
    else
        cout << "Deque c1 is greater than deque c2." << endl;
}
/* Output:
Deque c1 is less than or equal to deque c2.
*/
```

operator==

Tests if the deque object on the left side of the operator is equal to the deque object on the right side.

```
bool operator==(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque` .

right

An object of type `deque` .

Return Value

true if the deque on the left side of the operator is equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deques are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// deque_op_eq.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1, c2;

    c1.push_back( 1 );
    c2.push_back( 1 );

    if ( c1 == c2 )
        cout << "The deques are equal." << endl;
    else
        cout << "The deques are not equal." << endl;

    c1.push_back( 1 );
    if ( c1 == c2 )
        cout << "The deques are equal." << endl;
    else
        cout << "The deques are not equal." << endl;
}
/* Output:
The deques are equal.
The deques are not equal.
*/
```

operator>

Tests if the deque object on the left side of the operator is greater than the deque object on the right side.

```
bool operator>(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Return Value

true if the deque on the left side of the operator is greater than the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// deque_op_gt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1, c2;

    c1.push_back( 1 );
    c1.push_back( 3 );
    c1.push_back( 1 );

    c2.push_back( 1 );
    c2.push_back( 2 );
    c2.push_back( 2 );

    if ( c1 > c2 )
        cout << "Deque c1 is greater than deque c2." << endl;
    else
        cout << "Deque c1 is not greater than deque c2." << endl;
}
/* Output:
Deque c1 is greater than deque c2.
*/

```

operator>=

Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

```
bool operator>=(const deque<Type, Allocator>& left, const deque<Type, Allocator>& right);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Return Value

true if the deque on the left side of the operator is greater than or equal to the deque on the right side of the operator; otherwise **false**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// deque_op_ge.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1, c2;

    c1.push_back( 1 );
    c1.push_back( 3 );
    c1.push_back( 1 );

    c2.push_back( 1 );
    c2.push_back( 2 );
    c2.push_back( 2 );

    if ( c1 >= c2 )
        cout << "Deque c1 is greater than or equal to deque c2." << endl;
    else
        cout << "Deque c1 is less than deque c2." << endl;
}
/* Output:
Deque c1 is greater than or equal to deque c2.
*/

```

See also

[<deque>](#)

deque Class

11/9/2018 • 40 minutes to read • [Edit Online](#)

Arranges elements of a given type in a linear arrangement and, like a vector, enables fast random access to any element, and efficient insertion and deletion at the back of the container. However, unlike a vector, the `deque` class also supports efficient insertion and deletion at the front of the container.

Syntax

```
template <class Type, class Allocator =allocator<Type>>
class deque
```

Parameters

Type

The element data type to be stored in the deque.

Allocator

The type that represents the stored allocator object that encapsulates details about the deque's allocation and deallocation of memory. This argument is optional, and the default value is **`allocator<Type>`**.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. [Vectors](#) should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the list container is superior when efficient insertions and deletions (in constant time) at any location within the sequence is at a premium. Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

Deque reallocation occurs when a member function must insert or erase elements of the sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by [begin](#) and [end](#) become invalid.
- If an element is inserted at the first position of the deque, then all iterators, but no references, that designate existing elements become invalid.
- If an element is inserted at the end of the deque, then [end](#) and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at the front of the deque, only that iterator and references to the erased element become invalid.
- If the last element is erased from the end of the deque, only that iterator to the final element and references to the erased element become invalid.

Otherwise, inserting or erasing an element invalidates all iterators and references.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>deque</code>	Constructs a <code>deque</code> . Several constructors are provided to set up the contents of the new <code>deque</code> in different ways: empty; loaded with a specified number of empty elements; contents moved or copied from another <code>deque</code> ; contents copied or moved by using an iterator; and one element copied into the <code>deque</code> <code>count</code> times. Some of the constructors enable using a custom <code>allocator</code> to create elements.

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for the <code>deque</code> object.
<code>const_iterator</code>	A type that provides a random-access iterator that can access and read elements in the <code>deque</code> as <code>const</code> .
<code>const_pointer</code>	A type that provides a pointer to an element in a <code>deque</code> as a <code>const</code> .
<code>const_reference</code>	A type that provides a reference to an element in a <code>deque</code> for reading and other operations as a <code>const</code> .
<code>const_reverse_iterator</code>	A type that provides a random-access iterator that can access and read elements in the <code>deque</code> as const . The deque is viewed in reverse. For more information, see reverse_iterator Class
<code>difference_type</code>	A type that provides the difference between two random-access iterators that refer to elements in the same <code>deque</code> .
<code>iterator</code>	A type that provides a random-access iterator that can read or modify any element in a <code>deque</code> .
<code>pointer</code>	A type that provides a pointer to an element in a <code>deque</code> .
<code>reference</code>	A type that provides a reference to an element stored in a <code>deque</code> .
<code>reverse_iterator</code>	A type that provides a random-access iterator that can read or modify an element in a <code>deque</code> . The deque is viewed in reverse order.
<code>size_type</code>	A type that counts the number of elements in a <code>deque</code> .
<code>value_type</code>	A type that represents the data type stored in a <code>deque</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Erases elements from a <code>deque</code> and copies a new sequence of elements to the target <code>deque</code> .
<code>at</code>	Returns a reference to the element at a specified location in the <code>deque</code> .
<code>back</code>	Returns a reference to the last element of the <code>deque</code> .
<code>begin</code>	Returns a random-access iterator addressing the first element in the <code>deque</code> .
<code>cbegin</code>	Returns a const iterator to the first element in the <code>deque</code> .
<code>cend</code>	Returns a random-access const iterator that points just beyond the end of the <code>deque</code> .
<code>clear</code>	Erases all the elements of a <code>deque</code> .
<code>crbegin</code>	Returns a random-access const iterator to the first element in a <code>deque</code> viewed in reverse order.
<code>crend</code>	Returns a random-access const iterator to the first element in a <code>deque</code> viewed in reverse order.
<code>emplace</code>	Inserts an element constructed in place into the <code>deque</code> at a specified position.
<code>emplace_back</code>	Adds an element constructed in place to the end of the <code>deque</code> .
<code>emplace_front</code>	Adds an element constructed in place to the start of the <code>deque</code> .
<code>empty</code>	Returns true if the <code>deque</code> contains zero elements, and false if it contains one or more elements.
<code>end</code>	Returns a random-access iterator that points just beyond the end of the <code>deque</code> .
<code>erase</code>	Removes an element or a range of elements in a <code>deque</code> from specified positions.
<code>front</code>	Returns a reference to the first element in a <code>deque</code> .
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object that is used to construct the <code>deque</code> .
<code>insert</code>	Inserts an element, several elements, or a range of elements into the <code>deque</code> at a specified position.
<code>max_size</code>	Returns the maximum possible length of the <code>deque</code> .

MEMBER FUNCTION	DESCRIPTION
pop_back	Erases the element at the end of the <code>deque</code> .
pop_front	Erases the element at the start of the <code>deque</code> .
push_back	Adds an element to the end of the <code>deque</code> .
push_front	Adds an element to the start of the <code>deque</code> .
rbegin	Returns a random-access iterator to the first element in a reversed <code>deque</code> .
rend	Returns a random-access iterator that points just beyond the last element in a reversed <code>deque</code> .
resize	Specifies a new size for a <code>deque</code> .
shrink_to_fit	Discards excess capacity.
size	Returns the number of elements in the <code>deque</code> .
swap	Exchanges the elements of two <code>deque</code> s.

Operators

OPERATOR	DESCRIPTION
operator[]	Returns a reference to the <code>deque</code> element at a specified position.
operator=	Replaces the elements of the <code>deque</code> with a copy of another <code>deque</code> .

Requirements

Header: `<deque>`

`deque::allocator_type`

A type that represents the allocator class for the deque object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

Example

See the example for [get_allocator](#).

`deque::assign`

Erases elements from a deque and copies a new set of elements to the target deque.

```
template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);

void assign(
    size_type Count,
    const Type& Val);

void assign(initializer_list<Type> IList);
```

Parameters

First

Position of the first element in the range of elements to be copied from the argument deque.

Last

Position of the first element beyond the range of elements to be copied from the argument deque.

Count

The number of copies of an element being inserted into the deque.

Val

The value of the element being inserted into the deque.

IList

The initializer_list being inserted into the deque.

Remarks

After any existing elements in the target deque are erased, `assign` either inserts a specified range of elements from the original deque or from some other deque into the target deque, or inserts copies of a new element of a specified value into the target deque.

Example

```

// deque_assign.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
#include <initializer_list>

int main()
{
    using namespace std;
    deque<int> c1, c2;
    deque<int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    deque<int> d1{ 1, 2, 3, 4 };
    initializer_list<int> ilist{ 5, 6, 7, 8 };
    d1.assign(ilist);

    cout << "d1 = ";
    for (int i : d1)
        cout << i;
    cout << endl;

    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(7, 4);
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;
}

```

```
d1 = 5678c1 =102030c1 =5060c1 =4444444
```

deque::at

Returns a reference to the element at a specified location in the deque.

```

reference at(size_type pos);

const_reference at(size_type pos) const;

```

Parameters

pos

The subscript (or position number) of the element to reference in the deque.

Return Value

If *pos* is greater than the size of the deque, `at` throws an exception.

Return Value

If the return value of `at` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `at` is assigned to a `reference`, the deque object can be modified.

Example

```
// deque_at.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );

    const int& i = c1.at( 0 );
    int& j = c1.at( 1 );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;
}
```

```
The first element is 10
The second element is 20
```

deque::back

Returns a reference to the last element of the deque.

```
reference back();
const_reference back() const;
```

Return Value

The last element of the deque. If the deque is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `back` is assigned to a `reference`, the deque object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty deque. See [Checked Iterators](#) for more information.

Example

```

// deque_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_back( 10 );
    c1.push_back( 11 );

    int& i = c1.back( );
    const int& ii = c1.front( );

    cout << "The last integer of c1 is " << i << endl;
    i--;
    cout << "The next-to-last integer of c1 is " << ii << endl;
}

```

```

The last integer of c1 is 11
The next-to-last integer of c1 is 10

```

deque::begin

Returns an iterator addressing the first element in the deque.

```

const_iterator begin() const;
iterator begin();

```

Return Value

A random-access iterator addressing the first element in the deque or to the location succeeding an empty deque.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the deque object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the deque object can be modified.

Example

```
// deque_begin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;
    deque<int>::iterator c1_Iter;
    deque<int>::const_iterator c1_cIter;

    c1.push_back( 1 );
    c1.push_back( 2 );

    c1_Iter = c1.begin( );
    cout << "The first element of c1 is " << *c1_Iter << endl;

    *c1_Iter = 20;
    c1_Iter = c1.begin( );
    cout << "The first element of c1 is now " << *c1_Iter << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}
```

```
The first element of c1 is 1
The first element of c1 is now 20
```

deque::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin` , the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator` . Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()` .

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

deque::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.


```
const_iterator cend() const;
```

Return Value

A random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

deque::clear

Erases all the elements of a deque.

```
void clear();
```

Example

```
// deque_clear.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    cout << "The size of the deque is initially " << c1.size( ) << endl;
    c1.clear( );
    cout << "The size of the deque after clearing is " << c1.size( ) << endl;
}
```

```
The size of the deque is initially 3
The size of the deque after clearing is 0
```

deque::const_iterator

A type that provides a random-access iterator that can access and read a **const** element in the deque.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for [back](#).

deque::const_pointer

Provides a pointer to a **const** element in a deque.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element. An [iterator](#) is more commonly used to access a deque element.

deque::const_reference

A type that provides a reference to a **const** element stored in a deque for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

Example

```
// deque_const_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );

    const deque<int> c2 = c1;
    const int &i = c2.front( );
    const int &j = c2.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

    // The following line would cause an error as c2 is const
    // c2.push_back( 30 );
}
```

```
The first element is 10
The second element is 20
```

deque::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the deque.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the deque in reverse.

Example

See the example for [rbegin](#) for an example of how to declare and use an iterator.

deque::crbegin

Returns a const iterator to the first element in a reversed deque.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse random-access iterator addressing the first element in a reversed [deque](#) or addressing what had been the last element in the unreversed `deque`.

Remarks

With the return value of `crbegin`, the `deque` object cannot be modified.

Example

```
// deque_crbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> v1;
    deque<int>::iterator v1_Iter;
    deque<int>::const_reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    v1_Iter = v1.begin( );
    cout << "The first element of deque is "
         << *v1_Iter << "." << endl;

    v1_rIter = v1.crbegin( );
    cout << "The first element of the reversed deque is "
         << *v1_rIter << "." << endl;
}
```

The first element of deque is 1.
The first element of the reversed deque is 2.

deque::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed deque.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse random-access iterator that addresses the location succeeding the last element in a reversed [deque](#) (the location that had preceded the first element in the unreversed deque).

Remarks

`crend` is used with a reversed `deque` just as [array::cend](#) is used with a `deque`.

With the return value of `crend` (suitably decremented), the `deque` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its deque.

The value returned by `crend` should not be dereferenced.

Example

```
// deque_crend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> v1;
    deque <int>::const_reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
        cout << *v1_rIter << endl;
}
```

```
2
1
```

deque::deque

Constructs a deque of a specific size, or with elements of a specific value, or with a specific allocator, or as a copy of all or part of some other deque.

```

deque();

explicit deque(const Allocator& Al);
explicit deque(size_type Count);
deque(size_type Count, const Type& Val);

deque(
    size_type Count,
    const Type& Val,
    const Allocator& Al);

deque(const deque& Right);

template <class InputIterator>
deque(InputIterator First, InputIterator Last);

template <class InputIterator>
deque(
    InputIterator First,
    InputIterator Last,
    const Allocator& Al);

deque(initializer_list<value_type> Ilist, const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The allocator class to use with this object.
<i>Count</i>	The number of elements in the constructed deque.
<i>Val</i>	The value of the elements in the constructed deque.
<i>Right</i>	The deque of which the constructed deque is to be a copy.
<i>First</i>	Position of the first element in the range of elements to be copied.
<i>Last</i>	Position of the first element beyond the range of elements to be copied.
<i>*Ilist`</i>	The initializer_list to be copied.

Remarks

All constructors store an allocator object (*Al*) and initialize the deque.

The first two constructors specify an empty initial deque; the second one also specifies the allocator type (`_Al`) to be used.

The third constructor specifies a repetition of a specified number (`count`) of elements of the default value for class `Type` .

The fourth and fifth constructors specify a repetition of (*Count*) elements of value `val` .

The sixth constructor specifies a copy of the deque *Right*.

The seventh and eighth constructors copy the range `[First, Last)` of a deque.

The seventh constructor moves the deque *Right*.

The eighth constructor copies the contents of an `initializer_list`.

None of the constructors perform any interim reallocations.

Example

```
/ compile with: /EHsc
#include <deque>
#include <iostream>
#include <forward_list>

int main()
{
    using namespace std;

    forward_list<int> f1{ 1, 2, 3, 4 };

    f1.insert_after(f1.begin(), { 5, 6, 7, 8 });

    deque<int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

    // Create an empty deque c0
    deque<int> c0;

    // Create a deque c1 with 3 elements of default value 0
    deque<int> c1(3);

    // Create a deque c2 with 5 elements of value 2
    deque<int> c2(5, 2);

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque<int> c3(3, 1, c2.get_allocator());

    // Create a copy, deque c4, of deque c2
    deque<int> c4(c2);

    // Create a deque c5 by copying the range c4[ first, last)
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    deque<int> c5(c4.begin(), c4_Iter);

    // Create a deque c6 by copying the range c4[ first, last) and
    // c2 with the allocator of deque
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    c4_Iter++;
    deque<int> c6(c4.begin(), c4_Iter, c2.get_allocator());

    // Create a deque c8 by copying the contents of an initializer_list
    // using brace initialization
    deque<int> c8({ 1, 2, 3, 4 });

    initializer_list<int> iList{ 5, 6, 7, 8 };
    deque<int> c9( iList);

    cout << "c1 = ";
    for (int i : c1)
        cout << i << " ";
    cout << endl;

    cout << "c2 = ";
    for (int i : c2)
        cout << i << " ";
    cout << endl;
```

```

    cout << "c3 = ";
    for (int i : c3)
        cout << i << " ";
    cout << endl;

    cout << "c4 = ";
    for (int i : c4)
        cout << i << " ";
    cout << endl;

    cout << "c5 = ";
    for (int i : c5)
        cout << i << " ";
    cout << endl;

    cout << "c6 = ";
    for (int i : c6)
        cout << i << " ";
    cout << endl;

    // Move deque c6 to deque c7
    deque<int> c7(move(c6));
    deque<int>::iterator c7_Iter;

    cout << "c7 =";
    for (int i : c7)
        cout << i << " ";
    cout << endl;

    cout << "c8 = ";
    for (int i : c8)
        cout << i << " ";
    cout << endl;

    cout << "c9 = ";
    for (int i : c9)
        cout << i << " ";
    cout << endl;

    int x = 3;
}
// deque_deque.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

    // Create an empty deque c0
    deque<int> c0;

    // Create a deque c1 with 3 elements of default value 0
    deque<int> c1( 3 );

    // Create a deque c2 with 5 elements of value 2
    deque<int> c2( 5, 2 );

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque<int> c3( 3, 1, c2.get_allocator( ) );

    // Create a copy, deque c4, of deque c2
    deque<int> c4( c2 );

    // Create a deque c5 by copying the range c4[ first, last)

```

```

c4_Iter = c4.begin( );
c4_Iter++;
c4_Iter++;
deque<int> c5( c4.begin( ), c4_Iter );

// Create a deque c6 by copying the range c4[ first, last) and
// c2 with the allocator of deque
c4_Iter = c4.begin( );
c4_Iter++;
c4_Iter++;
c4_Iter++;
deque<int> c6( c4.begin( ), c4_Iter, c2.get_allocator( ) );

// Create a deque c8 by copying the contents of an initializer_list
// using brace initialization
deque<int> c8({ 1, 2, 3, 4 });

    initializer_list<int> ilist{ 5, 6, 7, 8 };
deque<int> c9( ilist);

cout << "c1 = ";
for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
    cout << *c1_Iter << " ";
cout << endl;

cout << "c2 = ";
for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
    cout << *c2_Iter << " ";
cout << endl;

cout << "c3 = ";
for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
    cout << *c3_Iter << " ";
cout << endl;

cout << "c4 = ";
for ( c4_Iter = c4.begin( ); c4_Iter != c4.end( ); c4_Iter++ )
    cout << *c4_Iter << " ";
cout << endl;

cout << "c5 = ";
for ( c5_Iter = c5.begin( ); c5_Iter != c5.end( ); c5_Iter++ )
    cout << *c5_Iter << " ";
cout << endl;

cout << "c6 = ";
for ( c6_Iter = c6.begin( ); c6_Iter != c6.end( ); c6_Iter++ )
    cout << *c6_Iter << " ";
cout << endl;

// Move deque c6 to deque c7
deque<int> c7( move(c6) );
deque<int>::iterator c7_Iter;

cout << "c7 = " ;
for ( c7_Iter = c7.begin( ); c7_Iter != c7.end( ); c7_Iter++ )
    cout << " " << *c7_Iter;
cout << endl;

cout << "c8 = ";
for (int i : c8)
    cout << i << " ";
cout << endl;

cout << "c9 = ";
or (int i : c9)
    cout << i << " ";
cout << endl;
}

```


deque::difference_type

A type that provides the difference between two iterators that refer to elements within the same deque.

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

A `difference_type` can also be described as the number of elements between two pointers.

Example

```
// deque_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <deque>
#include <algorithm>

int main( )
{
    using namespace std;

    deque<int> c1;
    deque<int>::iterator c1_Iter, c2_Iter;

    c1.push_back( 30 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1.push_back( 10 );
    c1.push_back( 30 );
    c1.push_back( 20 );

    c1_Iter = c1.begin( );
    c2_Iter = c1.end( );

    deque<int>::difference_type df_typ1, df_typ2, df_typ3;

    df_typ1 = count( c1_Iter, c2_Iter, 10 );
    df_typ2 = count( c1_Iter, c2_Iter, 20 );
    df_typ3 = count( c1_Iter, c2_Iter, 30 );
    cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
    cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
    cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

deque::emplace

Inserts an element constructed in place into the deque at a specified position.

```
iterator emplace(
    const_iterator _Where,
    Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Where</code>	The position in the <code>deque</code> where the first element is inserted.
<code>val</code>	The value of the element being inserted into the <code>deque</code> .

Return Value

The function returns an iterator that points to the position where the new element was inserted into the deque.

Remarks

Any insertion operation can be expensive, see `deque` for a discussion of `deque` performance.

Example

```
// deque_emplace.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> v1;
    deque<int>::iterator Iter;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );

    cout << "v1 =" ;
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    // initialize a deque of deques by moving v1
    deque< deque<int> > vv1;

    vv1.emplace( vv1.begin(), move( v1 ) );
    if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
    {
        cout << "vv1[0] =";
        for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
            cout << " " << *Iter;
        cout << endl;
    }
}
```

```
v1 = 10 20 30
vv1[0] = 10 20 30
```

deque::emplace_back

Adds an element constructed in place to the end of the deque.

```
void emplace_back(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the end of the deque .

Example

```
// deque_emplace_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> v1;

    v1.push_back( 1 );
    if ( v1.size( ) != 0 )
        cout << "Last element: " << v1.back( ) << endl;

    v1.push_back( 2 );
    if ( v1.size( ) != 0 )
        cout << "New last element: " << v1.back( ) << endl;

    // initialize a deque of deques by moving v1
    deque< deque<int> > vv1;

    vv1.emplace_back( move( v1 ) );
    if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
        cout << "Moved last element: " << vv1[0].back( ) << endl;
}
```

```
Last element: 1
New last element: 2
Moved last element: 2
```

deque::emplace_front

Adds an element constructed in place to the end of the deque.

```
void emplace_front(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the deque .

Example

```

// deque_emplace_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> v1;

    v1.push_back( 1 );
    if ( v1.size( ) != 0 )
        cout << "Last element: " << v1.back( ) << endl;

    v1.push_back( 2 );
    if ( v1.size( ) != 0 )
        cout << "New last element: " << v1.back( ) << endl;

    // initialize a deque of deques by moving v1
    deque< deque<int> > vv1;

    vv1.emplace_front( move( v1 ) );
    if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
        cout << "Moved last element: " << vv1[0].back( ) << endl;
}

```

```

Last element: 1
New last element: 2
Moved last element: 2

```

deque::empty

Tests if a deque is empty.

```
bool empty() const;
```

Return Value

true if the deque is empty; **false** if the deque is not empty.

Example

```

// deque_empty.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 10 );
    if ( c1.empty( ) )
        cout << "The deque is empty." << endl;
    else
        cout << "The deque is not empty." << endl;
}

```

The deque is not empty.

deque::end

Returns an iterator that addresses the location succeeding the last element in a deque.

```
const_iterator end() const;

iterator end();
```

Return Value

A random-access iterator that addresses the location succeeding the last element in a deque. If the deque is empty, then `deque::end == deque::begin`.

Remarks

`end` is used to test whether an iterator has reached the end of its deque.

Example

```
// deque_end.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;
    deque<int>::iterator c1_Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_Iter = c1.end( );
    c1_Iter--;
    cout << "The last integer of c1 is " << *c1_Iter << endl;

    c1_Iter--;
    *c1_Iter = 400;
    cout << "The new next-to-last integer of c1 is " << *c1_Iter << endl;

    // If a const iterator had been declared instead with the line:
    // deque<int>::const_iterator c1_Iter;
    // an error would have resulted when inserting the 400

    cout << "The deque is now:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
}
```

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The deque is now: 10 400 30
```

deque::erase

Removes an element or a range of elements in a deque from specified positions.

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);
```

Parameters

_Where

Position of the element to be removed from the deque.

first

Position of the first element removed from the deque.

last

Position just beyond the last element removed from the deque.

Return Value

A random-access iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the deque if no such element exists.

Remarks

`erase` never throws an exception.

Example

```
// deque_erase.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;
    deque <int>::iterator Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1.push_back( 40 );
    c1.push_back( 50 );
    cout << "The initial deque is: ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << *Iter << " ";
    cout << endl;
    c1.erase( c1.begin( ) );
    cout << "After erasing the first element, the deque becomes: ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << *Iter << " ";
    cout << endl;
    Iter = c1.begin( );
    Iter++;
    c1.erase( Iter, c1.end( ) );
    cout << "After erasing all elements but the first, deque becomes: ";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << *Iter << " ";
    cout << endl;
}
```

```
The initial deque is: 10 20 30 40 50
After erasing the first element, the deque becomes: 20 30 40 50
After erasing all elements but the first, deque becomes: 20
```

deque::front

Returns a reference to the first element in a deque.

```
reference front();

const_reference front() const;
```

Return Value

If the deque is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `front` is assigned to a `reference`, the deque object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty deque. See [Checked Iterators](#) for more information.

Example

```
// deque_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_back( 10 );
    c1.push_back( 11 );

    int& i = c1.front( );
    const int& ii = c1.front( );

    cout << "The first integer of c1 is " << i << endl;
    i++;
    cout << "The second integer of c1 is " << ii << endl;
}
```

```
The first integer of c1 is 10
The second integer of c1 is 11
```

deque::get_allocator

Returns a copy of the allocator object used to construct the deque.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the deque.

Remarks

Allocators for the deque class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// deque_get_allocator.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    // The following lines declare objects that use the default allocator.
    deque<int> c1;
    deque<int, allocator<int> > c2 = deque<int, allocator<int> >( allocator<int>( ) );

    // c3 will use the same allocator class as c1
    deque<int> c3( c1.get_allocator( ) );

    deque<int>::allocator_type xlst = c1.get_allocator( );
    // You can now call functions on the allocator class used by c1
}
```

deque::insert

Inserts an element or a number of elements or a range of elements into the deque at a specified position.

```
iterator insert(
    const_iterator Where,
    const Type& Val);

iterator insert(
    const_iterator Where,
    Type&& Val);

void insert(
    iterator Where,
    size_type Count,
    const Type& Val);

template <class InputIterator>
void insert(
    iterator Where,
    InputIterator First,
    InputIterator Last);

iterator insert(
    iterator Where,initializer_list<Type>
    Ilist);
```

Parameters

PARAMETER	DESCRIPTION
Where	The position in the target deque where the first element is inserted.

PARAMETER	DESCRIPTION
<i>Val</i>	The value of the element being inserted into the deque.
<i>Count</i>	The number of elements being inserted into the deque.
<i>First</i>	The position of the first element in the range of elements in the argument deque to be copied.
<i>Last</i>	The position of the first element beyond the range of elements in the argument deque to be copied.
<i>IList</i>	The initializer_list of elements to insert.

Return Value

The first two insert functions return an iterator that points to the position where the new element was inserted into the deque.

Remarks

Any insertion operation can be expensive.

deque::iterator

A type that provides a random-access iterator that can read or modify any element in a deque.

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for [begin](#).

deque::max_size

Returns the maximum length of the deque.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the deque.

Example

```
// deque_max_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;
    deque <int>::size_type i;

    i = c1.max_size( );
    cout << "The maximum possible length of the deque is " << i << "." << endl;
}
```

deque::operator[]

Returns a reference to the deque element at a specified position.

```
reference operator[](size_type pos);

const_reference operator[](size_type pos) const;
```

Parameters

pos

The position of the deque element to be referenced.

Return Value

A reference to the element whose position is specified in the argument. If the position specified is greater than the size of the deque, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `operator[]` is assigned to a `reference`, the deque object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the deque. See [Checked Iterators](#) for more information.

Example

```
// deque_op_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );
    cout << "The first integer of c1 is " << c1[0] << endl;
    int& i = c1[1];
    cout << "The second integer of c1 is " << i << endl;
}
```

```
The first integer of c1 is 10
The second integer of c1 is 20
```

deque::operator=

Replaces the elements of this deque using the elements from another deque.

```
deque& operator=(const deque& right);

deque& operator=(deque&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The deque that provides the new content.

Remarks

The first override copies elements to this deque from *right*, the source of the assignment. The second override moves elements to this deque from *right*.

Elements that are contained in this deque before the operator executes are removed.

Example

```

// deque_operator_as.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
using namespace std;

typedef deque<int> MyDeque;

template<typename MyDeque> struct S;

template<typename MyDeque> struct S<MyDeque&> {
    static void show( MyDeque& d ) {
        MyDeque::const_iterator iter;
        for (iter = d.cbegin(); iter != d.cend(); iter++)
            cout << *iter << " ";
        cout << endl;
    }
};

template<typename MyDeque> struct S<MyDeque&&> {
    static void show( MyDeque&& d ) {
        MyDeque::const_iterator iter;
        for (iter = d.cbegin(); iter != d.cend(); iter++)
            cout << *iter << " ";
        cout << " via unnamed rvalue reference " << endl;
    }
};

int main( )
{
    MyDeque d1, d2;

    d1.push_back(10);
    d1.push_back(20);
    d1.push_back(30);
    d1.push_back(40);
    d1.push_back(50);

    cout << "d1 = " ;
    S<MyDeque&>::show( d1 );

    d2 = d1;
    cout << "d2 = ";
    S<MyDeque&>::show( d2 );

    cout << "      ";
    S<MyDeque&&>::show ( move< MyDeque& > (d1) );
}

```

deque::pointer

Provides a pointer to an element in a [deque](#).

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element. An [iterator](#) is more commonly used to access a deque element.

deque::pop_back

Deletes the element at the end of the deque.

```
void pop_back();
```

Remarks

The last element must not be empty. `pop_back` never throws an exception.

Example

```
// deque_pop_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 1 );
    c1.push_back( 2 );
    cout << "The first element is: " << c1.front( ) << endl;
    cout << "The last element is: " << c1.back( ) << endl;

    c1.pop_back( );
    cout << "After deleting the element at the end of the deque, the "
         << "last element is: " << c1.back( ) << endl;
}
```

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the deque, the last element is: 1
```

deque::pop_front

Deletes the element at the beginning of the deque.

```
void pop_front();
```

Remarks

The first element must not be empty. `pop_front` never throws an exception.

Example

```
// deque_pop_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_back( 1 );
    c1.push_back( 2 );
    cout << "The first element is: " << c1.front( ) << endl;
    cout << "The second element is: " << c1.back( ) << endl;

    c1.pop_front( );
    cout << "After deleting the element at the beginning of the "
        "deque, the first element is: " << c1.front( ) << endl;
}
```

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the deque, the first element is: 2
```

deque::push_back

Adds an element to the end of the deque.

```
void push_back(const Type& val);

void push_back(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the end of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

deque::push_front

Adds an element to the beginning of the deque.

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

Example

```
// deque_push_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_front( 1 );
    if ( c1.size( ) != 0 )
        cout << "First element: " << c1.front( ) << endl;

    c1.push_front( 2 );
    if ( c1.size( ) != 0 )
        cout << "New first element: " << c1.front( ) << endl;

    // move initialize a deque of strings
    deque <string> c2;
    string str("a");

    c2.push_front( move( str ) );
    cout << "Moved first element: " << c2.front( ) << endl;
}
```

```
First element: 1
New first element: 2
Moved first element: a
```

deque::rbegin

Returns an iterator to the first element in a reversed deque.

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse random-access iterator addressing the first element in a reversed deque or addressing what had been the last element in the unreversed deque.

Remarks

`rbegin` is used with a reversed deque just as `begin` is used with a deque.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the deque object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the deque object can be modified.

`rbegin` can be used to iterate through a deque backwards.

Example

```

// deque_rbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;
    deque<int>::iterator c1_Iter;
    deque<int>::reverse_iterator c1_rIter;

    // If the following line had replaced the line above, an error
    // would have resulted in the line modifying an element
    // (commented below) because the iterator would have been const
    // deque<int>::const_reverse_iterator c1_rIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_rIter = c1.rbegin( );
    cout << "Last element in the deque is " << *c1_rIter << "." << endl;

    cout << "The deque contains the elements: ";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << *c1_Iter << " ";
    cout << "in that order.";
    cout << endl;

    // rbegin can be used to iterate through a deque in reverse order
    cout << "The reversed deque is: ";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << *c1_rIter << " ";
    cout << endl;

    c1_rIter = c1.rbegin( );
    *c1_rIter = 40; // This would have caused an error if a
                   // const_reverse iterator had been declared as
                   // noted above
    cout << "Last element in deque is now " << *c1_rIter << "." << endl;
}

```

```

Last element in the deque is 30.
The deque contains the elements: 10 20 30 in that order.
The reversed deque is: 30 20 10
Last element in deque is now 40.

```

deque::reference

A type that provides a reference to an element stored in a deque.

```

typedef typename Allocator::reference reference;

```

Example


```
// deque_reference.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque <int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );

    const int &i = c1.front( );
    int &j = c1.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;
}
```

```
The first element is 10
The second element is 20
```

deque::rend

Returns an iterator that addresses the location succeeding the last element in a reversed deque.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed deque (the location that had preceded the first element in the unreversed deque).

Remarks

`rend` is used with a reversed deque just as `end` is used with a deque.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the deque object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, the deque object can be modified.

`rend` can be used to test whether a reverse iterator has reached the end of its deque.

The value returned by `rend` should not be dereferenced.

Example

```

// deque_rend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;

    deque<int> c1;
    deque<int>::iterator c1_Iter;
    deque<int>::reverse_iterator c1_rIter;
    // If the following line had replaced the line above, an error
    // would have resulted in the line modifying an element
    // (commented below) because the iterator would have been const
    // deque<int>::const_reverse_iterator c1_rIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_rIter = c1.rend( );
    c1_rIter --; // Decrementing a reverse iterator moves it forward
                 // in the deque (to point to the first element here)
    cout << "The first element in the deque is: " << *c1_rIter << endl;

    cout << "The deque is: ";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << *c1_Iter << " ";
    cout << endl;

    // rend can be used to test if an iteration is through all of
    // the elements of a reversed deque
    cout << "The reversed deque is: ";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << *c1_rIter << " ";
    cout << endl;

    c1_rIter = c1.rend( );
    c1_rIter--; // Decrementing the reverse iterator moves it backward
                // in the reversed deque (to the last element here)
    *c1_rIter = 40; // This modification of the last element would
                    // have caused an error if a const_reverse
                    // iterator had been declared (as noted above)
    cout << "The modified reversed deque is: ";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << *c1_rIter << " ";
    cout << endl;
}

```

```

The first element in the deque is: 10
The deque is: 10 20 30
The reversed deque is: 30 20 10
The modified reversed deque is: 30 20 40

```

deque::resize

Specifies a new size for a deque.

```

void resize(size_type _Newsize);

void resize(size_type _Newsize, Type val);

```

Parameters

_Newsize

The new size of the deque.

val

The value of the new elements to be added to the deque if the new size is larger than the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the deque's size is less than the requested size, *_Newsize*, elements are added to the deque until it reaches the requested size.

If the deque's size is larger than the requested size, the elements closest to the end of the deque are deleted until the deque reaches the size *_Newsize*.

If the present size of the deque is the same as the requested size, no action is taken.

[size](#) reflects the current size of the deque.

Example

```
// deque_resize.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1.resize( 4,40 );
    cout << "The size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is " << c1.back( ) << endl;

    c1.resize( 5 );
    cout << "The size of c1 is now: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;

    c1.resize( 2 );
    cout << "The reduced size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

```
The size of c1 is: 4
The value of the last element is 40
The size of c1 is now: 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

deque::reverse_iterator

A type that provides a random-access iterator that can read or modify an element in a reversed deque.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the deque.

Example

See the example for `rbegin`.

deque::shrink_to_fit

Discards excess capacity.

```
void shrink_to_fit();
```

Remarks

There is no portable way to determine if `shrink_to_fit` reduces the storage used by a [deque](#).

Example

```
// deque_shrink_to_fit.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> v1;
    //deque<int>::iterator Iter;

    v1.push_back( 1 );
    v1.push_back( 2 );
    cout << "Current size of v1 = "
         << v1.size( ) << endl;
    v1.shrink_to_fit();
    cout << "Current size of v1 = "
         << v1.size( ) << endl;
}
```

```
Current size of v1 = 1
Current size of v1 = 1
```

deque::size

Returns the number of elements in the deque.

```
size_type size() const;
```

Return Value

The current length of the deque.

Example

```
// deque_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1;
    deque<int>::size_type i;

    c1.push_back( 1 );
    i = c1.size( );
    cout << "The deque length is " << i << "." << endl;

    c1.push_back( 2 );
    i = c1.size( );
    cout << "The deque length is now " << i << "." << endl;
}
```

```
The deque length is 1.
The deque length is now 2.
```

deque::size_type

A type that counts the number of elements in a deque.

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for [size](#).

deque::swap

Exchanges the elements of two deques.

```
void swap(deque<Type, Allocator>& right);

friend void swap(deque<Type, Allocator>& left, deque<Type, Allocator>& right) template <class Type, class
Allocator>
void swap(deque<Type, Allocator>& left, deque<Type, Allocator>& right);
```

Parameters

right

The deque providing the elements to be swapped, or the deque whose elements are to be exchanged with those of the deque `left`.

left

A deque whose elements are to be exchanged with those of the deque *right*.

Example

```

// deque_swap.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
    deque<int> c1, c2, c3;
    deque<int>::iterator c1_Iter;

    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 3 );
    c2.push_back( 10 );
    c2.push_back( 20 );
    c3.push_back( 100 );

    cout << "The original deque c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    c1.swap( c2 );

    cout << "After swapping with c2, deque c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    swap( c1,c3 );

    cout << "After swapping with c3, deque c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    swap<>(c1, c2);
    cout << "After swapping with c2, deque c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;
}

```

```

The original deque c1 is: 1 2 3
After swapping with c2, deque c1 is: 10 20
After swapping with c3, deque c1 is: 100
After swapping with c2, deque c1 is: 1 2 3

```

deque::value_type

A type that represents the data type stored in a deque.

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Type`.

Example

```
// deque_value_type.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
int main( )
{
    using namespace std;
    deque<int>::value_type AnInt;
    AnInt = 44;
    cout << AnInt << endl;
}
```

44

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<exception>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines several types and functions related to the handling of exceptions. Exception handling is used in situations in which the system can recover from an error. It provides a means for control to be returned from a function to the program. The objective of incorporating exception handling is to increase the program's robustness while providing a way to recover from an error in an orderly fashion.

Syntax

```
#include <exception>
```

Typedefs

TYPE NAME	DESCRIPTION
exception_ptr	A type that describes a pointer to an exception.
terminate_handler	A type that describes a pointer to a function suitable for use as a <code>terminate_handler</code> .
unexpected_handler	A type that describes a pointer to a function suitable for use as an <code>unexpected_handler</code> .

Functions

FUNCTION	DESCRIPTION
current_exception	Obtains a pointer to the current exception.
get_terminate	Obtains the current <code>terminate_handler</code> function.
get_unexpected	Obtains the current <code>unexpected_handler</code> function.
make_exception_ptr	Creates an <code>exception_ptr</code> object that holds a copy of an exception.
rethrow_exception	Throws an exception passed as a parameter.
set_terminate	Establishes a new <code>terminate_handler</code> to be called at the termination of the program.
set_unexpected	Establishes a new <code>unexpected_handler</code> to be when an unexpected exception is encountered.
terminate	Calls a terminate handler.
uncaught_exception	Returns true only if a thrown exception is being currently processed.

FUNCTION	DESCRIPTION
unexpected	Calls an unexpected handler.

Classes

CLASS	DESCRIPTION
bad_exception Class	The class describes an exception that can be thrown from an <code>unexpected_handler</code> .
exception Class	The class serves as the base class for all exceptions thrown by certain expressions and by the C++ Standard Library.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<exception> functions

11/9/2018 • 5 minutes to read • [Edit Online](#)

current_exception	get_terminate	get_unexpected
make_exception_ptr	rethrow_exception	set_terminate
set_unexpected	terminate	uncaught_exception
unexpected		

current_exception

Obtains a smart pointer to the current exception.

```
exception_ptr current_exception();
```

Return Value

An [exception_ptr](#) object pointing to the current exception.

Remarks

Call the `current_exception` function in a catch block. If an exception is in flight and the catch block can catch the exception, the `current_exception` function returns an `exception_ptr` object that references the exception. Otherwise, the function returns a null `exception_ptr` object.

The `current_exception` function captures the exception that is in flight regardless of whether the **catch** statement specifies an [exception-declaration](#) statement.

The destructor for the current exception is called at the end of the **catch** block if you do not rethrow the exception. However, even if you call the `current_exception` function in the destructor, the function returns an `exception_ptr` object that references the current exception.

Successive calls to the `current_exception` function return `exception_ptr` objects that refer to different copies of the current exception. Consequently, the objects compare as unequal because they refer to different copies, even though the copies have the same binary value.

make_exception_ptr

Creates an [exception_ptr](#) object that holds a copy of an exception.

```
template <class E>  
exception_ptr make_exception_ptr(E Except);
```

Parameters

Except

The class with the exception to copy. Usually, you specify an [exception class](#) object as the argument to the `make_exception_ptr` function, although any class object can be the argument.

Return Value

An [exception_ptr](#) object pointing to a copy of the current exception for *Except*.

Remarks

Calling the `make_exception_ptr` function is equivalent to throwing a C++ exception, catching it in a catch block, and then calling the [current_exception](#) function to return an `exception_ptr` object that references the exception. The Microsoft implementation of the `make_exception_ptr` function is more efficient than throwing and then catching an exception.

An application typically does not require the `make_exception_ptr` function, and we discourage its use.

rethrow_exception

Throws an exception passed as a parameter.

```
void rethrow_exception(exception_ptr P);
```

Parameters

P

The caught exception to re-throw. If *P* is a null [exception_ptr](#), the function throws [std::bad_exception](#).

Remarks

After you store a caught exception in an `exception_ptr` object, the primary thread can process the object. In your primary thread, call the `rethrow_exception` function together with the `exception_ptr` object as its argument. The `rethrow_exception` function extracts the exception from the `exception_ptr` object and then throws the exception in the context of the primary thread.

get_terminate

Obtains the current `terminate_handler` function.

```
terminate_handler get_terminate();
```

set_terminate

Establishes a new `terminate_handler` to be called at the termination of the program.

```
terminate_handler set_terminate(terminate_handler fnew) throw();
```

Parameters

fnew

The function to be called at termination.

Return Value

The address of the previous function that used to be called at termination.

Remarks

The function establishes a new [terminate_handler](#) as the function * *fnew*. Thus, *fnew* must not be a null pointer. The function returns the address of the previous terminate handler.

Example

```

// exception_set_terminate.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>

using namespace std;

void termfunction()
{
    cout << "My terminate function called." << endl;
    abort();
}

int main()
{
    terminate_handler oldHandler = set_terminate(termfunction);

    // Throwing an unhandled exception would also terminate the program
    // or we could explicitly call terminate();

    //throw bad_alloc();
    terminate();
}

```

get_unexpected

Obtains the current `unexpected_handler` function.

```
unexpected_handler get_unexpected();
```

set_unexpected

Establishes a new `unexpected_handler` to be when an unexpected exception is encountered.

```
unexpected_handler set_unexpected(unexpected_handler fnew) throw();
```

Parameters

fnew

The function to be called when an unexpected exception is encountered.

Return Value

The address of the previous `unexpected_handler`.

Remarks

fnew must not be a null pointer.

The C++ Standard requires that `unexpected` is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The following example calls `unexpected` directly, which then calls the `unexpected_handler`.

Example

```
// exception_set_unexpected.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>

using namespace std;

void uefunction()
{
    cout << "My unhandled exception function called." << endl;
    terminate(); // this is what unexpected() calls by default
}

int main()
{
    unexpected_handler oldHandler = set_unexpected(uefunction);

    unexpected(); // library function to force calling the
                // current unexpected handler
}
```

terminate

Calls a terminate handler.

```
void terminate();
```

Remarks

The function calls a terminate handler, a function of type **void**. If `terminate` is called directly by the program, the terminate handler is the one most recently set by a call to [set_terminate](#). If `terminate` is called for any of several other reasons during evaluation of a throw expression, the terminate handler is the one in effect immediately after evaluating the throw expression.

A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls `abort`.

Example

See [set_unexpected](#) for an example of the use of `terminate`.

uncaught_exception

Returns **true** only if a thrown exception is being currently processed.

```
bool uncaught_exception();
```

Return Value

Returns **true** after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling [unexpected](#) as a result of the throw expression. In particular, `uncaught_exception` will return **true** when called from a destructor that is being invoked during an exception unwind. On devices, `uncaught_exception` is only supported on Windows CE 5.00 and higher versions, including Windows Mobile 2005 platforms.

Example

```

// exception_uncaught_exception.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <string>

class Test
{
public:
    Test( std::string msg ) : m_msg( msg )
    {
        std::cout << "In Test::Test(\"" << m_msg << "\"" << std::endl;
    }
    ~Test( )
    {
        std::cout << "In Test::~Test(\"" << m_msg << "\"" << std::endl
        << "        std::uncaught_exception( ) = "
        << std::uncaught_exception( )
        << std::endl;
    }
private:
    std::string m_msg;
};

// uncaught_exception will be true in the destructor
// for the object created inside the try block because
// the destructor is being called as part of the unwind.

int main( void )
{
    Test t1( "outside try block" );
    try
    {
        Test t2( "inside try block" );
        throw 1;
    }
    catch (...) {
    }
}

```

```

In Test::Test("outside try block")
In Test::Test("inside try block")
In Test::~Test("inside try block")
    std::uncaught_exception( ) = 1
In Test::~Test("outside try block")
    std::uncaught_exception( ) = 0

```

unexpected

Calls the unexpected handler.

```
void unexpected();
```

Remarks

The C++ Standard requires that `unexpected` is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The example calls `unexpected` directly, which calls the unexpected handler.

The function calls an unexpected handler, a function of type **void**. If `unexpected` is called directly by the program, the unexpected handler is the one most recently set by a call to [set_unexpected](#).

An unexpected handler may not return to its caller. It may terminate execution by:

- Throwing an object of a type listed in the exception specification or an object of any type if the unexpected handler is called directly by the program.
- Throwing an object of type [bad_exception](#).
- Calling [terminate](#), `abort` or `exit(int)`.

At program startup, the unexpected handler is a function that calls [terminate](#).

Example

See [set_unexpected](#) for an example of the use of `unexpected`.

See also

[<exception>](#)

<exception> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

exception_ptr	terminate_handler	unexpected_handler

exception_ptr

A type that describes a pointer to an exception.

```
typedef unspecified exception_ptr;
```

Remarks

An unspecified internal class that is used to implement the `exception_ptr` type.

Use an `exception_ptr` object to reference the current exception or an instance of a user-specified exception. In the Microsoft implementation, an exception is represented by an `EXCEPTION_RECORD` structure. Each `exception_ptr` object includes an exception reference field that points to a copy of the `EXCEPTION_RECORD` structure that represents the exception.

When you declare an `exception_ptr` variable, the variable is not associated with any exception. That is, its exception reference field is NULL. Such an `exception_ptr` object is called a *null exception_ptr*.

Use the `current_exception` or `make_exception_ptr` function to assign an exception to an `exception_ptr` object. When you assign an exception to an `exception_ptr` variable, the variable's exception reference field points to a copy of the exception. If there is insufficient memory to copy the exception, the exception reference field points to a copy of a `std::bad_alloc` exception. If the `current_exception` or `make_exception_ptr` function cannot copy the exception for any other reason, the function calls the `terminate` CRT function to exit the current process.

Despite its name, an `exception_ptr` object is not itself a pointer. It does not obey pointer semantics and cannot be used with the pointer member access (`->`) or indirection (`*`) operators. The `exception_ptr` object has no public data members or member functions.

Comparisons:

You can use the equal (`==`) and not-equal (`!=`) operators to compare two `exception_ptr` objects. The operators do not compare the binary value (bit pattern) of the `EXCEPTION_RECORD` structures that represent the exceptions. Instead, the operators compare the addresses in the exception reference field of the `exception_ptr` objects. Consequently, a null `exception_ptr` and the NULL value compare as equal.

terminate_handler

The type describes a pointer to a function suitable for use as a `terminate_handler` .

```
typedef void (*terminate_handler)();
```

Remarks

The type describes a pointer to a function suitable for use as a terminate handler.

Example

See [set_terminate](#) for an example of the use of `terminate_handler`.

unexpected_handler

The type describes a pointer to a function suitable for use as an `unexpected_handler`.

```
typedef void (*unexpected_handler)();
```

Example

See [set_unexpected](#) for an example of the use of `unexpected_handler`.

See also

[<exception>](#)

bad_exception Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class describes an exception that can be thrown from an unexpected handler.

Syntax

```
class bad_exception : public exception {};
```

Remarks

[unexpected](#) will throw a `bad_exception` instead of terminating or instead of calling another function specified with [set_unexpected](#) if `bad_exception` is included in the throw list of a function.

The value returned by `what` is an implementation-defined C string. None of the member functions throw any exceptions.

For a list of members inherited by the `bad_exception` class, see [exception Class](#).

Example

See [set_unexpected](#) for an example of the use of [unexpected](#) throwing a `bad_exception`.

Requirements

Header: <exception>

Namespace: std

See also

[exception Class](#)

[Thread Safety in the C++ Standard Library](#)

exception Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown by certain expressions and by the C++ Standard Library.

Syntax

```
class exception {
public:
    exception();
    exception(const char* const &message);
    exception(const char* const &message, int);
    exception(const exception &right);
    exception& operator=(const exception &right);
    virtual ~exception();
    virtual const char *what() const;
};
```

Remarks

Specifically, this base class is the root of the standard exception classes defined in [<stdexcept>](#). The C string value returned by `what` is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string. None of the member functions throw any exceptions.

The **int** parameter allows you to specify that no memory should be allocated. The value of the **int** is ignored.

NOTE

The constructors `exception(const char* const &message)` and `exception(const char* const &message, int)` are Microsoft extensions to the C++ Standard Library.

Example

For examples of the use of the standard exception classes that inherit from the `exception` class, see any of the classes defined in [<stdexcept>](#).

Requirements

Header: `<exception>`

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

<filesystem>

5/7/2019 • 3 minutes to read • [Edit Online](#)

Include the header <filesystem> for access to classes and functions that manipulate and retrieve information about paths, files and directories.

Syntax

```
#include <experimental/filesystem> // C++-standard header file name
#include <filesystem> // Microsoft-specific implementation header file name
using namespace std::experimental::filesystem::v1;
```

IMPORTANT

As of the release of Visual Studio 2017, the <filesystem> header was not yet a C++ standard. C++ in Visual Studio 2017 (MSVC v141) implements the final draft standard, found in [ISO/IEC JTC 1/SC 22/WG 21 N4100](#).

This header supports filesystems for one of two broad classes of host operating systems: Microsoft Windows and Posix.

While most functionality is common to both operating systems, this document identifies where differences occur. For example:

- Windows supports multiple root names, such as c: or \\network_name. A filesystem consists of a forest of trees, each with its own root directory, such as c:\ or \\network_name\, and each with its own current directory, for completing a relative pathname (one that is not an absolute pathname).
- Posix supports a single tree, with no root name, the single root directory /, and a single current directory.

Another significant difference is the native representation of pathnames:

- Windows uses a null-terminated sequence of wchar_t, encoded as UTF-16 (one or two elements for each character).
- Posix uses a null-terminated sequence of char, encoded as UTF-8 (one or more elements for each character).
- An object of class path stores the pathname in native form, but supports easy conversion between this stored form and several external forms:
 - A null-terminated sequence of char, encoded as favored by the operating system.
 - A null-terminated sequence of char, encoded as UTF-8.
 - A null-terminated sequence of wchar_t, encoded as favored by the operating system.
 - A null-terminated sequence of char16_t, encoded as UTF-16.
 - A null-terminated sequence of char32_t, encoded as UTF-32.

Interconversions between these representations are mediated, as needed, by the use of one or more `codecvt` facets. If a specific locale object is not designated, these facets are obtained from the global locale.

Another difference is the detail with which each operating system lets you specify file or directory access

permissions:

1. Windows records whether a file is read only or writable, an attribute that has no meaning for directories.
2. Posix records whether a file can be read, written, or executed (scanned if a directory), by the owner, by the owner's group, or by everybody, plus a few other permissions.

Common to both systems is the structure imposed on a pathname once you get past the root name. For the pathname `c:/abc/xyz/def.ext`:

- The root name is `c:`.
- The root directory is `/`.
- The root path is `c:/`.
- The relative path is `abc/xyz/def.ext`.
- The parent path is `c:/abc/xyz`.
- The filename is `def.ext`.
- The stem is `def`.
- The extension is `.ext`.

A minor difference is the **preferred separator**, between the sequence of directories in a pathname. Both operating systems let you write a forward slash `/`, but in some contexts Windows prefers a backslash `\`.

Finally, an important feature of path objects is that you can use them wherever a filename argument is required in the classes defined in the header `<fstream>`.

For more information and code examples, see [File System Navigation \(C++\)](#).

Classes

NAME	DESCRIPTION
directory_entry Class	Describes an object that is returned by a <code>directory_iterator</code> or a <code>recursive_directory_iterator</code> and contains a path.
directory_iterator Class	Describes an input iterator that sequences through the file names in a file-system directory.
filesystem_error Class	A base class for exceptions that are thrown to report a low-level system overflow.
path Class	Defines a class that stores an object of template type <code>String</code> that is suitable for use as a file name.
recursive_directory_iterator Class	Describes an input iterator that sequences through the file names in a file-system directory. The iterator can also descend into subdirectories.
file_status Class	Wraps a <code>file_type</code> .

Structs

NAME	DESCRIPTION
space_info Structure	Holds information about a volume.

Functions

[<filesystem> functions](#)

Operators

[<filesystem> operators](#)

Enumerations

NAME	DESCRIPTION
copy_options	An enumeration that is used with copy_file and determines behavior if a destination file already exists.
copy_options	An enumeration that is used with copy_file and determines behavior if a destination file already exists.
directory_options	An enumeration that specifies options for directory iterators.
file_type	An enumeration for file types.
perms	A bitmask type used to convey permissions and options to permissions

See also

[Header Files Reference](#)

<filesystem> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

The operators perform a lexical comparison of two paths as strings. Use the `equivalent` function to determine whether two paths (for example a relative path and an absolute path) refer to the same file or directory on disk.

For more information, see [File System Navigation \(C++\)](#).

operator==

```
bool operator==(const path& left, const path& right) noexcept;
```

The function returns `left.native() == right.native()`.

operator!=

```
bool operator!=(const path& left, const path& right) noexcept;
```

The function returns `!(left == right)`.

operator<

```
bool operator<(const path& left, const path& right) noexcept;
```

The function returns `left.native() < right.native()`.

operator<=

```
bool operator<=(const path& left, const path& right) noexcept;
```

The function returns `!(right < left)`.

operator>

```
bool operator>(const path& left, const path& right) noexcept;
```

The function returns `right < left`.

operator>=

```
bool operator>=(const path& left, const path& right) noexcept;
```

The function returns `!(left < right)`.

operator/

```
path operator/(const path& left, const path& right);
```

The function executes:

```
basic_string<Elem, Traits> str;  
path ans = left;  
return (ans /= right);
```

operator<<

```
template <class Elem, class Traits>  
basic_ostream<Elem, Traits>& operator<<(basic_ostream<Elem, Traits>& os, const path& pval);
```

The function returns `os << pval.string<Elem, Traits>()`.

operator>>

```
template <class Elem, class Traits>  
basic_istream<Elem, Traits>& operator>>(basic_istream<Elem, Traits>& is, const path& pval);
```

The function executes:

```
basic_string<Elem, Traits> str;  
is>> str;  
pval = str;  
return (is);
```

See also

[path Class \(C++ Standard Library\)](#)

[File System Navigation \(C++\)](#)

[<filesystem>](#)

<filesystem> functions

3/28/2019 • 9 minutes to read • [Edit Online](#)

These free functions in the [<filesystem>](#) header perform modifying and query operations on paths, files, symlinks, directories and volumes. For more information and code examples, see [File System Navigation \(C++\)](#).

absolute	begin	canonical
copy	copy_file	copy_symlink
create_directories	create_directory	create_directory_symlink
create_hard_link	create_symlink	current_path
end	equivalent	exists
file_size	hard_link_count	hash_value
is_block_file	is_character_file	is_directory
is_empty	is_fifo	is_other
is_regular_file	is_socket	is_symlink
last_write_time	permissions	read_symlink
remove	remove_all	rename
resize_file	space	status
status_known	swap	symlink_status
system_complete	temp_directory_path	u8path

absolute

```
path absolute(const path& pval, const path& base = current_path());
```

The function returns the absolute pathname corresponding to *pval* relative to the pathname `base`:

1. If `pval.has_root_name() && pval.has_root_directory()` the function returns *pval*.
2. If `pval.has_root_name() && !pval.has_root_directory()` the function returns `pval.root_name() / absolute(base).root_directory() / absolute(base).relative_path() / pval.relative_path()`.
3. If `!pval.has_root_name() && pval.has_root_directory()` the function returns `absolute(base).root_name() / pval`.

4. If `!pval.has_root_name() && !pval.has_root_directory()` the function returns `absolute(base) / pval`.

begin

```
const directory_iterator& begin(const directory_iterator& iter) noexcept;
const recursive_directory_iterator&
    begin(const recursive_directory_iterator& iter) noexcept;
```

Both functions return *iter*.

canonical

```
path canonical(const path& pval, const path& base = current_path());
path canonical(const path& pval, error_code& ec);
path canonical(const path& pval, const path& base, error_code& ec);
```

The functions all form an absolute pathname `pabs = absolute(pval, base)` (or `pabs = absolute(pval)` for the overload with no base parameter), then reduce it to a canonical form in the following sequence of steps:

1. Every path component `x` for which `is_symlink(X)` is **true** is replaced by `read_symlink(X)`.
2. Every path component `.` (dot is the current directory established by previous path components) is removed.
3. Every pair of path components `x / ..` (dot-dot is the parent directory established by previous path components) is removed.

The function then returns `pabs`.

copy

```
void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
void copy(const path& from, const path& to, copy_options opts);
void copy(const path& from, const path& to, copy_options opts, error_code& ec) noexcept;
```

The functions all possibly copy or link one or more files at *from* to *to* under control of *opts*, which is taken as `copy_options::none` for the overloads with no *opts* parameter. *opts* shall contain at most one of:

- `skip_existing`, `overwrite_existing`, OR `update_existing`
- `copy_symlinks` OR `skip_symlinks`
- `directories_only`, `create_symlinks`, OR `create_hard_links`

The functions first determine the file_status values `f` for *from* and `t` for *to*:

- if `opts & (copy_options::create_symlinks | copy_options::skip_symlinks)`, by calling `symlink_status`
- otherwise, by calling `status`
- Otherwise report an error.

If `!exists(f) || equivalent(f, t) || is_other(f) || is_other(t) || is_directory(f) && is_regular_file(t)`, they then report an error (and do nothing else).

Otherwise, if `is_symlink(f)` then:

- If `options & copy_options::skip_symlinks` then do nothing.
- Otherwise, if `!exists(t) && options & copy_options::copy_symlinks` then `copy_symlink(from, to, opts)`.
- Otherwise report an error.

Otherwise, if `is_regular_file(f)` then:

- If `opts & copy_options::directories_only` then do nothing.
- Otherwise, if `opts & copy_options::create_symlinks` then `create_symlink(to, from)`.
- Otherwise, if `opts & copy_options::create_hard_links` then `create_hard_link(to, from)`.
- Otherwise, if `is_directory(f)` then `copy_file(from, to / from.filename(), opts)`.
- Otherwise, `copy_file(from, to, opts)`.

Otherwise, if `is_directory(f) && (opts & copy_options::recursive || !opts)` then:

```
if (!exists(t))
{ // copy directory contents recursively
    create_directory(to, from, ec);

    for (directory_iterator next(from), end; ec == error_code() && next != end; ++next)
    {
        copy(next->path(), to / next->path().filename(), opts, ec);
    }
}
```

Otherwise, do nothing.

copy_file

```
bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec) noexcept;
bool copy_file(const path& from, const path& to, copy_options opts);
bool copy_file(const path& from, const path& to, copy_options opts, error_code& ec) noexcept;
```

The functions all possibly copy the file at *from* to *to* under control of *opts*, which is taken as `copy_options::none` for the overloads with no *opts* parameter. *opts* shall contain at most one of `skip_existing`, `overwrite_existing`, or `update_existing`.

If

```
exists(to) && !(opts & (copy_options::skip_existing | copy_options::overwrite_existing |
copy_options::update_existing))
```

then report as an error that the file already exists.

Otherwise, if

```
!exists(to) || opts & copy_options::overwrite_existing || opts & copy_options::update_existing &&
last_write_time(to) < last_write_time(from) || !(opts & (copy_options::skip_existing |
copy_options::overwrite_existing | copy_options::update_existing))
```

then attempt to copy the contents and attributes of the file *from* to the file *to*. Report as an error if the copy attempt fails.

The functions return **true** if the copy is attempted and succeeds, otherwise **false**.

copy_symlink

```
void copy_symlink(const path& from, const path& to);  
void copy_symlink(const path& from, const path& to, error_code& ec) noexcept;
```

If `is_directory(from)` the function calls `create_directory_symlink(from, to)`. Otherwise, it calls `create_symlink(from, to)`.

create_directories

```
bool create_directories(const path& pval);  
bool create_directories(const path& pval, error_code& ec) noexcept;
```

For a pathname such as `a/b/c` the function creates directories `a` and `a/b` as needed so that it can create the directory `a/b/c` as needed. It returns **true** only if it actually creates the directory `pval`.

create_directory

```
bool create_directory(const path& pval);  
  
bool create_directory(const path& pval, error_code& ec) noexcept;  
bool create_directory(const path& pval, const path& attr);  
bool create_directory(const path& pval, const path& attr, error_code& ec) noexcept;
```

The function creates the directory `pval` as needed. It returns true only if it actually creates the directory `pval`, in which case it copies permissions from the existing file `attr`, or uses `perms::all` for the overloads with no `attr` parameter.

create_directory_symlink

```
void create_directory_symlink(const path& to, const path& link);  
void create_directory_symlink(const path& to, const path& link, error_code& ec) noexcept;
```

The function creates `link` as a symlink to the directory `to`.

create_hard_link

```
void create_hard_link(const path& to, const path& link);  
void create_hard_link(const path& to, const path& link, error_code& ec) noexcept;
```

The function creates `link` as a hard link to the directory or file `to`.

create_symlink

```
void create_symlink(const path& to, const path& link);  
  
void create_symlink(const path& to, const path& link, error_code& ec) noexcept;
```

The function creates `link` as a symlink to the file `to`.

current_path

```
path current_path();
path current_path(error_code& ec);
void current_path(const path& pval);
void current_path(const path& pval, error_code& ec) noexcept;
```

The functions with no parameter *pval* return the pathname for the current directory. The remaining functions set the current directory to *pval*.

end

```
directory_iterator& end(const directory_iterator& iter) noexcept;
recursive_directory_iterator& end(const recursive_directory_iterator& iter) noexcept;
```

The first function returns `directory_iterator()` and the second function returns `recursive_directory_iterator()`

equivalent

```
bool equivalent(const path& left, const path& right);
bool equivalent(const path& left, const path& right, error_code& ec) noexcept;
```

The functions return **true** only if *left* and *right* designate the same filesystem entity.

exists

```
bool exists(file_status stat) noexcept;
bool exists(const path& pval);
bool exists(const path& pval, error_code& ec) noexcept;
```

The first function returns `status_known && stat.type() != file_not_found`. The second and third functions return `exists(status(pval))`.

file_size

```
uintmax_t file_size(const path& pval);
uintmax_t file_size(const path& pval, error_code& ec) noexcept;
```

The functions return the size in bytes of the file designated by *pval*, if `exists(pval) && is_regular_file(pval)` and the file size can be determined. Otherwise they report an error and return `uintmax_t(-1)`.

hard_link_count

```
uintmax_t hard_link_count(const path& pval);
uintmax_t hard_link_count(const path& pval, error_code& ec) noexcept;
```

The function returns the number of hard links for *pval*, or -1 if an error occurs.

hash_value

```
size_t hash_value(const path& pval) noexcept;
```

The function returns a hash value for `pval.native()` .

is_block_file

```
bool is_block_file(file_status stat) noexcept;  
bool is_block_file(const path& pval);  
bool is_block_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::block` . The remaining functions return `is_block_file(status(pval))` .

is_character_file

```
bool is_character_file(file_status stat) noexcept;  
bool is_character_file(const path& pval);  
bool is_character_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::character` . The remaining functions return `is_character_file(status(pval))` .

is_directory

```
bool is_directory(file_status stat) noexcept;  
bool is_directory(const path& pval);  
bool is_directory(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::directory` . The remaining functions return `is_directory_file(status(pval))` .

is_empty

```
bool is_empty(file_status stat) noexcept;  
bool is_empty(const path& pval);  
bool is_empty(const path& pval, error_code& ec) noexcept;
```

If `is_directory(pval)` then the function returns `directory_iterator(pval) == directory_iterator()` ; otherwise it returns `file_size(pval) == 0` .

is_fifo

```
bool is_fifo(file_status stat) noexcept;  
bool is_fifo(const path& pval);  
bool is_fifo(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::fifo` . The remaining functions return `is_fifo(status(pval))` .

is_other

```
bool is_other(file_status stat) noexcept;
bool is_other(const path& pval);
bool is_other(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::other`. The remaining functions return `is_other(status(pval))`.

is_regular_file

```
bool is_regular_file(file_status stat) noexcept;
bool is_regular_file(const path& pval);
bool is_regular_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::regular`. The remaining functions return `is_regular_file(status(pval))`.

is_socket

```
bool is_socket(file_status stat) noexcept;
bool is_socket(const path& pval);
bool is_socket(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::socket`. The remaining functions return `is_socket(status(pval))`.

is_symlink

```
bool is_symlink(file_status stat) noexcept;
bool is_symlink(const path& pval);
bool is_symlink(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::symlink`. The remaining functions return `is_symlink(status(pval))`.

last_write_time

```
file_time_type last_write_time(const path& pval);
file_time_type last_write_time(const path& pval, error_code& ec) noexcept;
void last_write_time(const path& pval, file_time_type new_time);
void last_write_time(const path& pval, file_time_type new_time, error_code& ec) noexcept;
```

The first two functions return the time of last data modification for *pval*, or `file_time_type(-1)` if an error occurs. The last two functions set the time of last data modification for *pval* to *new_time*.

permissions

```
void permissions(const path& pval, perms mask);
void permissions(const path& pval, perms mask, error_code& ec) noexcept;
```

The functions set the permissions for the pathname designated by *pval* to `mask & perms::mask` under control of

`perms & (perms::add_perms | perms::remove_perms)` . *mask* shall contain at most one of `perms::add_perms` and `perms::remove_perms` .

If `mask & perms::add_perms` the functions set the permissions to `status(pval).permissions() | mask & perms::mask` . Otherwise, if `mask & perms::remove_perms` the functions set the permissions to `status(pval).permissions() & ~(mask & perms::mask)` . Otherwise, the functions set the permissions to `mask & perms::mask` .

read_symlink

```
path read_symlink(const path& pval);  
path read_symlink(const path& pval, error_code& ec);
```

The functions report an error and return `path()` if `!is_symlink(pval)` . Otherwise, the functions return an object of type `path` containing the symbolic link.

remove

```
bool remove(const path& pval);  
bool remove(const path& pval, error_code& ec) noexcept;
```

The functions return **true** only if `exists(symlink_status(pval))` and the file is successfully removed. A symlink is itself removed, not the file it designates.

remove_all

```
uintmax_t remove_all(const path& pval);  
uintmax_t remove_all(const path& pval, error_code& ec) noexcept;
```

If *pval* is a directory, the functions recursively remove all directory entries, then the entry itself. Otherwise, the functions call `remove` . They return a count of all elements successfully removed.

rename

```
void rename(const path& from, const path& to);  
void rename(const path& from, const path& to, error_code& ec) noexcept;
```

The functions rename *from* to *to* . A symlink is itself renamed, not the file it designates.

resize_file

```
void resize(const path& pval, uintmax_t size);  
void resize(const path& pval, uintmax_t size, error_code& ec) noexcept;
```

The functions alter the size of a file such that `file_size(pval) == size`

space


```
space_info space(const path& pval);
space_info space(const path& pval, error_code& ec) noexcept;
```

The function returns information about the volume designated by *pval*, in a structure of type `space_info`. The structure contains `uintmax_t(-1)` for any value that cannot be determined.

status

```
file_status status(const path& pval);
file_status status(const path& pval, error_code& ec) noexcept;
```

The functions return the pathname status, the file type and permissions, associated with *pval*. A symlink is itself not tested, but the file it designates.

status_known

```
bool status_known(file_status stat) noexcept;
```

The function returns `stat.type() != file_type::none`

swap

```
void swap(path& left, path& right) noexcept;
```

The function exchanges the contents of *left* and *right*.

symlink_status

```
file_status symlink_status(const path& pval);
file_status symlink_status(const path& pval, error_code& ec) noexcept;
```

The functions return the pathname symlink status, the file type and permissions, associated with *pval*. The functions behave the same as `status(pval)` except that a symlink is itself tested, not the file it designates.

system_complete

```
path system_complete(const path& pval);
path system_complete(const path& pval, error_code& ec);
```

The functions return an absolute pathname that takes into account, as necessary, the current directory associated with its root name. (For Posix, the functions return `absolute(pval)`.)

temp_directory_path

```
path temp_directory_path();
path temp_directory_path(error_code& ec);
```

The functions return a pathname for a directory suitable for containing temporary files.

u8path

```
template <class Source>
path u8path(const Source& source);

template <class InIt>
path u8path(InIt first, InIt last);
```

The first function behaves the same as `path(source)` and the second function behaves the same as `path(first, last)` except that the designated source in each case is taken as a sequence of char elements encoded as UTF-8, regardless of the filesystem.

<filesystem> enumerations

10/31/2018 • 2 minutes to read • [Edit Online](#)

This topic documents the enums in the filesystem header.

Requirements

Header: <experimental/filesystem>

Namespace: std::experimental::filesystem

copy_options

An enumeration of bitmask values that is used with [copy](#) and [copy_file](#) functions to specify behavior.

Syntax

```
enum class copy_options {
    none = 0,
    skip_existing = 1,
    overwrite_existing = 2,
    update_existing = 4,
    recursive = 8,
    copy_symlinks = 16,
    skip_symlinks = 32,
    directories_only = 64,
    create_symlinks = 128,
    create_hard_links = 256
};
```

Values

NAME	DESCRIPTION
<code>none</code>	Perform the default behavior for the operation.
<code>skip_existing</code>	Do not copy if the file already exists, do not report an error.
<code>overwrite_existing</code>	Overwrite the file if it already exists.
<code>update_existing</code>	Overwrite the file if it already exists and is older than the replacement.
<code>recursive</code>	Recursively copy subdirectories and their contents.
<code>copy_symlinks</code>	Copy symbolic links as symbolic links, instead of copying the files they point to.
<code>skip_symlinks</code>	Ignore symbolic links.
<code>directories_only</code>	Only iterate over directories, ignore files.

NAME	DESCRIPTION
<code>create_symlinks</code>	Make symbolic links instead of copying files. An absolute path must be used as the source path unless the destination is the current directory.
<code>create_hard_links</code>	Make hard links instead of copying files.

directory_options

Specifies whether to follow symbolic links to directories or to ignore them.

Syntax

```
enum class directory_options {
    none = 0,
    follow_directory_symlink
};
```

Values

NAME	DESCRIPTION
<code>none</code>	Default behavior: ignore symbolic links to directories. Permission denied is an error.
<code>follow_directory_symlink</code>	Treat symbolic links to directories as actual directories.

file_type

An enumeration for file types. The supported values are regular, directory, not_found, and unknown.

Syntax

```
enum class file_type {
    not_found = -1,
    none,
    regular,
    directory,
    symlink,
    block,
    character,
    fifo,
    socket,
    unknown
};
```

Values

NAME	VALUE	DESCRIPTION
<code>not_found</code>	-1	Represents a file that does not exist.
<code>none</code>	0	Represents a file that has no type attribute. (Not supported.)

NAME	VALUE	DESCRIPTION
<code>regular</code>	1	Represents a conventional disk file.
<code>directory</code>	2	Represents a directory.
<code>symlink</code>	3	Represents a symbolic link. (Not supported.)
<code>block</code>	4	Represents a block-special file on UNIX-based systems. (Not supported.)
<code>character</code>	5	Represents a character-special file on UNIX-based systems. (Not supported.)
<code>fifo</code>	6	Represents a FIFO file on UNIX-based systems. (Not supported.)
<code>socket</code>	7	Represents a socket on UNIX based systems. (Not supported.)
<code>unknown</code>	8	Represents a file whose status cannot be determined.

perms

Flags for file permissions. The supported values are essentially “readonly” and all. For a readonly file, none of the *_write bits are set. Otherwise the `all` bit (0x0777) is set.

Syntax

```
enum class perms { // names for permissions
    none = 0,
    owner_read = 0400, // S_IRUSR
    owner_write = 0200, // S_IWUSR
    owner_exec = 0100, // S_IXUSR
    owner_all = 0700, // S_IRWXU
    group_read = 040, // S_IRGRP
    group_write = 020, // S_IWGRP
    group_exec = 010, // S_IXGRP
    group_all = 070, // S_IRWXG
    others_read = 04, // S_IROTH
    others_write = 02, // S_IWOTH
    others_exec = 01, // S_IXOTH
    others_all = 07, // S_IRWXO
    all = 0777,
    set_uid = 04000, // S_ISUID
    set_gid = 02000, // S_ISGID
    sticky_bit = 01000, // S_ISVTX
    mask = 07777,
    unknown = 0xFFFF,
    add_perms = 0x10000,
    remove_perms = 0x20000,
    resolve_symlinks = 0x40000
};
```

See also

directory_entry Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an object that is returned by `*x`, where *X* is a [directory_iterator](#) or a [recursive_directory_iterator](#).

Syntax

```
class directory_entry;
```

Remarks

The class stores an object of type [path](#). The stored `path` can be an instance of the [path Class](#) or of a type that is derived from `path`. It also stores two [file_type](#) values; one that represents what is known about the status of the stored file name, and another that represents what is known about the symbolic link status of the file name.

For more information and code examples, see [File System Navigation \(C++\)](#).

Constructors

CONSTRUCTOR	DESCRIPTION
directory_entry	The defaulted constructors behave as expected. The fourth constructor initializes <code>mypath</code> to <i>pval</i> , <code>mystat</code> to <i>stat_arg</i> , and <code>mysymstat</code> to <i>symstat_arg</i> .

Member functions

MEMBER FUNCTION	DESCRIPTION
assign	The member function assigns <i>pval</i> to <code>mypath</code> , <i>stat</i> to <code>mystat</code> , and <i>symstat</i> to <code>mysymstat</code> .
path	The member function returns <code>mypath</code> .
replace_filename	The member function replaces <code>mypath</code> with <code>mypath.parent_path() / pval</code> , <code>mystat</code> with <i>stat_arg</i> , and <code>mysymstat</code> with <i>symstat_arg</i> .
status	Both member functions return <code>mystat</code> possibly first altered.
symlink_status	Both member functions return <code>mysymstat</code> possibly first altered.

Operators

OPERATOR	DESCRIPTION
operator!=	Replaces the elements of the list with a copy of another list.

OPERATOR	DESCRIPTION
<code>operator=</code>	The defaulted member assignment operators behave as expected.
<code>operator==</code>	Returns <code>mypath == right.mypath</code> .
<code>operator<</code>	Returns <code>mypath < right.mypath</code> .
<code>operator<=</code>	Returns <code>!(right < *this)</code> .
<code>operator></code>	Returns <code>right < *this</code> .
<code>operator>=</code>	Returns <code>!(*this < right)</code> .
<code>operator const path_type&</code>	Returns <code>mypath</code> .

Requirements

Header: `<experimental/filesystem>`

Namespace: `std::experimental::filesystem`

assign

The member function assigns *pval* to `mypath` , *stat_arg* to `mystat` , and *symstat_arg* to `mysymstat` .

```
void assign(const std::experimental::filesystem::path& pval,
            file_status stat_arg = file_status(),
            file_status symstat_arg = file_status());
```

Parameters

pval

The stored file name path.

stat_arg

The status of the stored file name.

symstat_arg

The symbolic link status of the stored file name.

directory_entry

The defaulted constructors behave as expected. The fourth constructor initializes `mypath` to *pval*, `mystat` to *stat_arg*, and `mysymstat` to *symstat_arg*.

```
directory_entry() = default;
directory_entry(const directory_entry&) = default;
directory_entry(directory_entry&&) noexcept = default;
explicit directory_entry(const std::experimental::filesystem::path& pval,
                        file_status stat_arg = file_status(),
                        file_status symstat_arg = file_status());
```


Parameters

pval

The stored file name path.

stat_arg

The status of the stored file name.

symstat_arg

The symbolic link status of the stored file name.

operator!=

The member function returns `!(*this == right)`.

```
bool operator!=(const directory_entry& right) const noexcept;
```

Parameters

right

The [directory_entry](#) being compared to the `directory_entry`.

operator=

The defaulted member assignment operators behave as expected.

```
directory_entry& operator=(const directory_entry&) = default;  
directory_entry& operator=(directory_entry&&) noexcept = default;
```

Parameters

right

The [directory_entry](#) being copied into the `directory_entry`.

operator==

The member function returns `mypath == right.mypath`.

```
bool operator==(const directory_entry& right) const noexcept;
```

Parameters

right

The [directory_entry](#) being compared to the `directory_entry`.

operator<

The member function returns `mypath < right.mypath`.

```
bool operator<(const directory_entry& right) const noexcept;
```

Parameters

right

The [directory_entry](#) being compared to the `directory_entry`.

operator<=

The member function returns `!(right < *this)` .

```
bool operator<=(const directory_entry& right) const noexcept;
```

Parameters

right

The `directory_entry` being compared to the `directory_entry` .

operator>

The member function returns `right < *this` .

```
bool operator>(const directory_entry& right) const noexcept;
```

Parameters

right

The `directory_entry` being compared to the `directory_entry` .

operator>=

The member function returns `!(*this < right)` .

```
bool operator>=(const directory_entry& right) const noexcept;
```

Parameters

right

The `directory_entry` being compared to the `directory_entry` .

operator const path_type&

The member operator returns `mypath` .

```
operator const std::experimental::filesystem::path&() const;
```

path

The member function returns `mypath` .

```
const std::experimental::filesystem::path& path() const noexcept;
```

replace_filename

The member function replaces `mypath` with `mypath.parent_path() / pval`, `mystat` with `stat_arg`, and `mysymstat` with `symstat_arg`

```
void replace_filename(  
    const std::experimental::filesystem::path& pval,  
    file_status stat_arg = file_status(),  
    file_status symstat_arg = file_status());
```

Parameters

pval

The stored file name path.

stat_arg

The status of the stored file name.

symstat_arg

The symbolic link status of the stored file name.

status

Both member functions return `mystat` possibly first altered as follows:

1. If `status_known(mystat)` then do nothing.
2. Otherwise, if `!status_known(mysymstat) && !is_symlink(mysymstat)` then `mystat = mysymstat`.

```
file_status status() const;  
file_status status(error_code& ec) const noexcept;
```

Parameters

ec

The status error code.

symlink_status

Both member functions return `mysymstat` possibly first altered as follows: If `status_known(mysymstat)` then do nothing. Otherwise, `mysymstat = symlink_status(mypval)`.

```
file_status symlink_status() const;  
file_status symlink_status(error_code& ec) const noexcept;
```

Parameters

ec

The status error code.

See also

[Header Files Reference](#)

[<filesystem>](#)

directory_iterator Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an input iterator that sequences through the filenames in a directory. For an iterator `x`, the expression `*x` evaluates to an object of class `directory_entry` that wraps the filename and anything known about its status.

The class stores an object of type `path`, called `mydir` here for the purposes of exposition, which represents the name of the directory to be sequenced, and an object of type `directory_entry` called `myentry` here, which represents the current filename in the directory sequence. A default constructed object of type `directory_iterator` has an empty `mydir` pathname and represents the end-of-sequence iterator.

For example, given the directory `abc` with entries `def` and `ghi`, the code:

```
for (directory_iterator next(path("abc")), end; next != end; ++next) visit(next->path());
```

will call `visit` with the arguments `path("abc/def")` and `path("abc/ghi")`.

For more information and code examples, see [File System Navigation \(C++\)](#).

Syntax

```
class directory_iterator;
```

Constructors

CONSTRUCTOR	DESCRIPTION
<code>directory_iterator</code>	Constructs an input iterator that sequences through the filenames in a directory.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>increment</code>	Attempts to advance to the next filename in the directory.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Returns <code>!(*this == right)</code> .
<code>operator=</code>	The defaulted member assignment operators behave as expected.
<code>operator==</code>	Returns true only if both <code>*this</code> and <code>right</code> are end-of-sequence iterators or both are not end-of-sequence-iterators.
<code>operator*</code>	Returns <code>myentry</code> .
<code>operator-></code>	Returns <code>&*this</code> .

OPERATOR	DESCRIPTION
<code>operator++</code>	Calls <code>increment()</code> , then returns <code>*this</code> , or makes a copy of the object, calls <code>increment()</code> , then returns the copy.

Requirements

Header: `<experimental/filesystem>`

Namespace: `std::experimental::filesystem`

`directory_iterator::directory_iterator`

The first constructor produces an end-of-sequence iterator. The second and third constructors store *pval* in `mydir`, then attempt to open and read `mydir` as a directory. If successful, they store the first filename in the directory in `myentry`; otherwise they produce an end-of-sequence iterator.

The defaulted constructors behave as expected.

```
directory_iterator() noexcept;
explicit directory_iterator(const path& pval);

directory_iterator(const path& pval, error_code& ec) noexcept;
directory_iterator(const directory_iterator&) = default;
directory_iterator(directory_iterator&&) noexcept = default;
```

Parameters

pval

The stored file name path.

ec

The status error code.

directory_iterator

The stored object.

`directory_iterator::increment`

The function attempts to advance to the next filename in the directory. If successful, it stores that filename in `myentry`; otherwise it produces an end-of-sequence iterator.

```
directory_iterator& increment(error_code& ec) noexcept;
```

`directory_iterator::operator!=`

The member operator returns `!(*this == right)`.

```
bool operator!=(const directory_iterator& right) const;
```

Parameters

right

The `directory_iterator` being compared to the `directory_iterator`.

`directory_iterator::operator=`

The defaulted member assignment operators behave as expected.

```
directory_iterator& operator=(const directory_iterator&) = default;  
directory_iterator& operator=(directory_iterator&&) noexcept = default;
```

Parameters

right

The `directory_iterator` being copied into the `directory_iterator`.

`directory_iterator::operator==`

The member operator returns **true** only if both `*this` and *right* are end-of-sequence iterators or both are not end-of-sequence-iterators.

```
bool operator==(const directory_iterator& right) const;
```

Parameters

right

The `directory_iterator` being compared to the `directory_iterator`.

`directory_iterator::operator*`

The member operator returns `myentry`.

```
const directory_entry& operator*() const;
```

`directory_iterator::operator->`

The member function returns `&*this`.

```
const directory_entry * operator->() const;
```

`directory_iterator::operator++`

The first member function calls `increment()`, then returns `*this`. The second member function makes a copy of the object, calls `increment()`, then returns the copy.

```
directory_iterator& operator++();  
directory_iterator& operator++(int);
```

Parameters

int

The number of increments.

See also

[Header Files Reference](#)

[<filesystem>](#)

[File System Navigation \(C++\)](#)

file_status Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Wraps a [file_type](#) and file [perms](#).

Syntax

```
class file_status;
```

Constructors

CONSTRUCTOR	DESCRIPTION
file_status	Constructs a wrapper for file_type and file perms .

Member functions

MEMBER FUNCTION	DESCRIPTION
type	Gets or sets the <code>file_type</code> .
permissions	Gets or sets the file permissions.

Operators

OPERATOR	DESCRIPTION
operator=	The defaulted member assignment operators behave as expected.

Requirements

Header: <filesystem>

Namespace: std::experimental::filesystem, std::experimental::filesystem

file_status::file_status

Constructs a wrapper for [file_type](#) and file [perms](#).

```
explicit file_status(
    file_type ftype = file_type::none,
    perms mask = perms::unknown) noexcept;

file_status(const file_status&) noexcept = default;

file_status(file_status&&) noexcept = default;

~file_status() noexcept = default;
```

Parameters

ftype

Specified `file_type`, defaults to `file_type::none`.

mask

Specified file `perms`, defaults to `perms::unknown`.

file_status

The stored object.

file_status::operator=

The defaulted member assignment operators behave as expected.

```
file_status& operator=(const file_status&) noexcept = default;
file_status& operator=(file_status&&) noexcept = default;
```

Parameters

file_status

The [file_status](#) being copied into the `file_status`.

type

Gets or sets the `file_type`.

```
file_type type() const noexcept
void type(file_type ftype) noexcept
```

Parameters

ftype

Specified `file_type`.

permissions

Gets or sets the file permissions.

Use the setter to make a file `readonly` or remove the `readonly` attribute.

```
perms permissions() const noexcept
void permissions(perms mask) noexcept
```

Parameters

mask

Specified `perms`.

See also

[Header Files Reference](#)

[path Class](#)

[<filesystem>](#)

filesystem_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A base class for all exceptions that are thrown to report a low-level system overflow.

Syntax

```
class filesystem_error : public system_error;
```

Remarks

The class serves as the base class for all exceptions thrown to report an error in <filesystem> functions. It stores an object of type `string`, called `mymesg` here for the purposes of exposition. It also stores two objects of type `path`, called `mypva11` and `mypva12`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>filesystem_error</code>	Constructs a <code>filesystem_error</code> message.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>path1</code>	Returns <code>mypva11</code>
<code>path2</code>	Returns <code>mypva12</code>
<code>what</code>	Returns a pointer to an <code>NTBS</code> .

Requirements

Header: <filesystem>

Namespace: `std::experimental::filesystem`

filesystem_error::filesystem_error

The first constructor constructs its message from *what_arg* and *ec*. The second constructor also constructs its message from *pva11*, which it stores in `mypva11`. The third constructor also constructs its message from *pva11*, which it stores in `mypva11`, and from *pva12*, which it stores in `mypva12`.

```

filesystem_error(const string& what_arg,
                error_code ec);

filesystem_error(const string& what_arg,
                const path& pval1,
                error_code ec);

filesystem_error(const string& what_arg,
                const path& pval1,
                const path& pval2,
                error_code ec);

```

Parameters

what_arg

Specified message.

ec

Specified error code.

mypval1

Further specified message parameter.

mypval2

Further specified message parameter.

filesystem_error::path1

The member function returns `mypval1`

```
const path& path1() const noexcept;
```

filesystem_error::path2

The member function returns `mypval2`

```
const path& path2() const noexcept;
```

filesystem_error::what

The member function returns a pointer to an `NTBS`, preferably composed from `runtime_error::what()`, `system_error::what()`, `mymesg`, `mypval1.native_string()`, and `mypval2.native_string()`.

```
const char *what() const noexcept;
```

See also

[Header Files Reference](#)

[system_error Class](#)

[<filesystem>](#)

[<exception>](#)

path Class

10/31/2018 • 10 minutes to read • [Edit Online](#)

The **path** class stores an object of type `string_type`, called `myname` here for the purposes of exposition, suitable for use as a pathname. `string_type` is a synonym for `basic_string<value_type>`, where `value_type` is a synonym for **wchar_t** on Windows or **char** on POSIX.

For more information, and code examples, see [File System Navigation \(C++\)](#).

Syntax

```
class path;
```

Constructors

CONSTRUCTOR	DESCRIPTION
path	Constructs a <code>path</code> .

Typedefs

TYPE NAME	DESCRIPTION
const_iterator	A synonym for <code>iterator</code> .
iterator	A bidirectional constant iterator that designates the <code>path</code> components of <code>myname</code> .
string_type	The type is a synonym for <code>basic_string<value_type></code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
append	Appends the specified sequence to <code>mypath</code> , converted and inserting a preferred_separator as needed.
assign	Replaces <code>mypath</code> with the specified sequence, converted as needed.
begin	Returns a <code>path::iterator</code> designating the first path element in the pathname, if present.
c_str	Returns a pointer to the first character in <code>mypath</code> .
clear	Executes <code>mypath.clear()</code> .
compare	Returns comparison values.

MEMBER FUNCTION	DESCRIPTION
<code>concat</code>	Appends the specified sequence to <code>mypath</code> , converted (but not inserting a separator) as needed.
<code>empty</code>	Returns <code>mypath.empty()</code> .
<code>end</code>	Returns an end-of-sequence iterator of type <code>iterator</code> .
<code>extension</code>	Returns the suffix of <code>filename()</code> .
<code>filename</code>	Returns the root directory component of myname, specifically <code>empty() path() : *--end()</code> . The component may be empty.
<code>generic_string</code>	Returns <code>this->string<Elem, Traits, Alloc>(al)</code> with (under Windows) any backslash converted to a forward slash.
<code>generic_u16string</code>	Returns <code>u16string()</code> with (under Windows) any backslash converted to a forward slash.
<code>generic_u32string</code>	Returns <code>u32string()</code> with (under Windows) any backslash converted to a forward slash.
<code>generic_u8string</code>	Returns <code>u8string()</code> with (under Windows) any backslash converted to a forward slash.
<code>generic_wstring</code>	Returns <code>wstring()</code> with (under Windows) any backslash converted to a forward slash.
<code>has_extension</code>	Returns <code>!extension().empty()</code> .
<code>has_filename</code>	Returns <code>!filename().empty()</code> .
<code>has_parent_path</code>	Returns <code>!parent_path().empty()</code> .
<code>has_relative_path</code>	Returns <code>!relative_path().empty()</code> .
<code>has_root_directory</code>	Returns <code>!root_directory().empty()</code> .
<code>has_root_name</code>	Returns <code>!root_name().empty()</code> .
<code>has_root_path</code>	Returns <code>!root_path().empty()</code> .
<code>has_stem</code>	Returns <code>!stem().empty()</code> .
<code>is_absolute</code>	For Windows, the function returns <code>has_root_name() && has_root_directory()</code> . For Posix, the function returns <code>has_root_directory()</code> .
<code>is_relative</code>	Returns <code>!is_absolute()</code> .

MEMBER FUNCTION	DESCRIPTION
<code>make_preferred</code>	Converts each separator to a preferred_separator as needed.
<code>native</code>	Returns <code>myname</code> .
<code>parent_path</code>	Returns the parent path component of <code>myname</code> .
<code>preferred_separator</code>	The constant object gives the preferred character for separating path components, depending on the host operating system.
<code>relative_path</code>	Returns the relative path component of <code>myname</code> .
<code>remove_filename</code>	Removes the filename.
<code>replace_extension</code>	Replaces the extension of <code>myname</code> .
<code>replace_filename</code>	Replaces the filename.
<code>root_directory</code>	Returns the root directory component of <code>myname</code> .
<code>root_name</code>	Returns the root name component of <code>myname</code> .
<code>root_path</code>	Returns the root path component of <code>myname</code> .
<code>stem</code>	Returns the <code>stem</code> component of <code>myname</code> .
<code>string</code>	Converts the sequence stored in <code>mypath</code> .
<code>swap</code>	Executes <code>swap(mypath, right.mypath)</code> .
<code>u16string</code>	Converts the sequence stored in <code>mypath</code> to UTF-16 and returns it stored in an object of type <code>u16string</code> .
<code>u32string</code>	Converts the sequence stored in <code>mypath</code> to UTF-32 and returns it stored in an object of type <code>u32string</code> .
<code>u8string</code>	Converts the sequence stored in <code>mypath</code> to UTF-8 and returns it stored in an object of type <code>u8string</code> .
<code>value_type</code>	The type describes the path elements favored by the host operating system.
<code>wstring</code>	Converts the sequence stored in <code>mypath</code> to the encoding favored by the host system for a <code>wchar_t</code> sequence and returns it stored in an object of type <code>wstring</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Replaces the elements of the path with a copy of another path.
<code>operator+=</code>	Various <code>concat</code> expressions.
<code>operator/=</code>	Various <code>append</code> expressions.
<code>operator string_type</code>	Returns <code>myname</code> .

Requirements

Header: `<filesystem>`

Namespace: `std::experimental::filesystem`

`path::append`

Appends the specified sequence to `mypath` , converted and inserting a `preferred_separator` as needed.

```
template <class Source>
path& append(const Source& source);

template <class InIt>
path& append(InIt first, InIt last);
```

Parameters

source

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

`path::assign`

Replaces `mypath` with the specified sequence, converted as needed.

```
template <class Source>
path& assign(const Source& source);

template <class InIt>
path& assign(InIt first, InIt last);
```

Parameters

source

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

path::begin

Returns a `path::iterator` designating the first path element in the pathname, if present.

```
iterator begin() const;
```

path::c_str

Returns a pointer to the first character in `mypath`.

```
const value_type& *c_str() const noexcept;
```

path::clear

Executes `mypath.clear()`.

```
void clear() noexcept;
```

path::compare

The first function returns `mypath.compare(pval.native())`. The second function returns `mypath.compare(str)`. The third function returns `mypath.compare(ptr)`.

```
int compare(const path& pval) const noexcept;  
int compare(const string_type& str) const;  
int compare(const value_type *ptr) const;
```

Parameters

pval

Path to compare.

str

String to compare.

ptr

Pointer to compare.

path::concat

Appends the specified sequence to `mypath`, converted (but not inserting a separator) as needed.

```
template <class Source>  
path& concat(const Source& source);  
  
template <class InIt>  
path& concat(InIt first, InIt last);
```

Parameters

source

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

path::const_iterator

A synonym for `iterator`.

```
typedef iterator const_iterator;
```

path::empty

Returns `mypath.empty()`.

```
bool empty() const noexcept;
```

path::end

Returns an end-of-sequence iterator of type `iterator`.

```
iterator end() const;
```

path::extension

Returns the suffix of `filename()`.

```
path extension() const;
```

Remarks

Returns the suffix of `filename() x` such that:

If `x == path(".")` || `x == path("..")` or if `x` contains no dot, the suffix is empty.

Otherwise, the suffix begins with (and includes) the rightmost dot.

path::filename

Returns the root directory component of myname, specifically `empty() path() : *--end()`. The component may be empty.

```
path filename() const;
```

path::generic_string

Returns `this->string<Elem, Traits, Alloc>(al)` with (under Windows) any backslash converted to a forward slash.

```
template <class Elem,
        class Traits = char_traits<Elem>,
        class Alloc = allocator<Elem>>
basic_string<Elem, Traits, Alloc>
    generic_string(const Alloc& al = Alloc()) const;

string generic_string() const;
```

path::generic_u16string

Returns `u16string()` with (under Windows) any backslash converted to a forward slash.

```
u16string generic_u16string() const;
```

path::generic_u32string

Returns `u32string()` with (under Windows) any backslash converted to a forward slash.

```
u32string generic_u32string() const;
```

path::generic_u8string

Returns `u8string()` with (under Windows) any backslash converted to a forward slash.

```
string generic_u8string() const;
```

path::generic_wstring

Returns `wstring()` with (under Windows) any backslash converted to a forward slash.

```
wstring generic_wstring() const;
```

path::has_extension

Returns `!extension().empty()` .

```
bool has_extension() const;
```

path::has_filename

Returns `!filename().empty()` .

```
bool has_filename() const;
```

path::has_parent_path

Returns `!parent_path().empty()` .

```
bool has_parent_path() const;
```

path::has_relative_path

Returns `!relative_path().empty()` .

```
bool has_relative_path() const;
```

path::has_root_directory

Returns `!root_directory().empty()` .

```
bool has_root_directory() const;
```

path::has_root_name

Returns `!root_name().empty()` .

```
bool has_root_name() const;
```

path::has_root_path

Returns `!root_path().empty()` .

```
bool has_root_path() const;
```

path::has_stem

Returns `!stem().empty()` .

```
bool has_stem() const;
```

path::is_absolute

For Windows, the function returns `has_root_name() && has_root_directory()` . For Posix, the function returns `has_root_directory()` .

```
bool is_absolute() const;
```

path::is_relative

Returns `!is_absolute()` .

```
bool is_relative() const;
```

path::iterator

A bidirectional constant iterator that designates the path components of `myname`.

```
class iterator
{
    // bidirectional iterator for path
    typedef bidirectional_iterator_tag iterator_category;
    typedef path_type value_type;
    typedef ptrdiff_t difference_type;
    typedef const value_type *pointer;
    typedef const value_type& reference;
    // ...
};
```

Remarks

The class describes a bidirectional constant iterator that designates the `path` components of `myname` in the sequence:

1. the root name, if present
2. the root directory, if present
3. the remaining directory elements of the parent `path`, if present, ending with the filename, if present

For `pval` an object of type `path`:

1. `path::iterator X = pval.begin()` designates the first `path` element in the pathname, if present.
2. `X == pval.end()` is true when `X` points just past the end of the sequence of components.
3. `*X` returns a string that matches the current component
4. `++X` designates the next component in the sequence, if present.
5. `--X` designates the preceding component in the sequence, if present.
6. Altering `myname` invalidates all iterators designating elements in `myname`.

path::make_preferred

Converts each separator to a `preferred_separator` as needed.

```
path& make_preferred();
```

path::native

Returns `myname`.

```
const string_type& native() const noexcept;
```

path::operator=

Replaces the elements of the path with a copy of another path.

```

path& operator=(const path& right);
path& operator=(path&& right) noexcept;

template <class Source>
path& operator=(const Source& source);

```

Parameters

right

The [path](#) being copied into the `path`.

source

The source path.

Remarks

The first member operator copies `right.myname` to `myname`. The second member operator moves `right.myname` to `myname`. The third member operator behaves the same as `*this = path(source)`.

path::operator+=

Various `concat` expressions.

```

path& operator+=(const path& right);
path& operator+=(const string_type& str);
path& operator+=(const value_type *ptr);
path& operator+=(value_type elem);

template <class Source>
path& operator+=(const Source& source);

template <class Elem>
path& operator+=(Elem elem);

```

Parameters

right

The added path.

str

The added string.

ptr

The added pointer.

elem

The added `value_type` or `Elem`.

source

The added source.

Remarks

The member functions behave the same as the following corresponding expressions:

1. `concat(right);`
2. `concat(path(str));`
3. `concat(ptr);`

4. `concat(string_type(1, elem));`
5. `concat(source);`
6. `concat(path(basic_string<Elem>(1, elem)));`

path::operator/=

Various `append` expressions.

```
path& operator/=(const path& right);

template <class Source>
path& operator/=(const Source& source);
```

Parameters

right

The added path.

source

The added source.

Remarks

The member functions behave the same as the following corresponding expressions:

1. `append(right);`
2. `append(source);`

path::operator string_type

Returns `myname` .

```
operator string_type() const;
```

path::parent_path

Returns the parent path component of `myname` .

```
path parent_path() const;
```

Remarks

Returns the parent path component of `myname` , specifically the prefix of `myname` after removing `filename().native()` and any immediately preceding directory separators. (Equally, if `begin() != end()` , it is the combining of all elements in the range `[begin(), --end())` by successively applying `operator/=` .) The component may be empty.

path::path

Constructs a `path` in various ways.

```

path();

path(const path& right);
path(path&& right) noexcept;

template <class Source>
path(const Source& source);

template <class Source>
path(const Source& source, const locale& loc);

template <class InIt>
path(InIt first, InIt last);

template <class InIt>
path(InIt first, InIt last, const locale& loc);

```

Parameters

right

The path of which the constructed path is to be a copy.

source

The source of which the constructed path is to be a copy.

loc

The specified locale.

first

The position of the first element to be copied.

last

The position of the last element to be copied.

Remarks

The constructors all construct `myname` in various ways:

For `path()` it is `myname()`.

For `path(const path& right)` it is `myname(right.myname)`.

For `path(path&& right)` it is `myname(right.myname)`.

For `template<class Source> path(const Source& source)` it is `myname(source)`.

For `template<class Source> path(const Source& source, const locale& loc)` it is `myname(source)`, obtaining any needed codecvt facets from `loc`.

For `template<class InIt> path(InIt first, InIt last)` it is `myname(first, last)`.

For `template<class InIt> path(InIt first, InIt last, const locale& loc)` it is `myname(first, last)`, obtaining any needed codecvt facets from `loc`.

path::preferred_separator

The constant object gives the preferred character for separating path components, depending on the host operating system.

```
#if _WIN32_C_LIB
static constexpr value_type preferred_separator == L'\\';
#else // assume Posix
static constexpr value_type preferred_separator == '/';
#endif // filesystem model now defined
```

Remarks

Note that it is equally permissible in most contexts under Windows to use L'/' in its place.

path::relative_path

Returns the relative path component of `myname`.

```
path relative_path() const;
```

Remarks

Returns the relative path component of `myname`, specifically the suffix of `myname` after removing `root_path().native()` and any immediately subsequent redundant directory separators. The component may be empty.

path::remove_filename

Removes the filename.

```
path& remove_filename();
```

path::replace_extension

Replaces the extension of `myname`.

```
path& replace_extension(const path& newext = path());
```

Parameters

newext

The new extension.

Remarks

First removes the suffix `extension().native()` from `myname`. Then if `!newext.empty() && newext[0] != dot` (where `dot` is `*path(".").c_str()`), then `dot` is appended to `myname`. Then *newext* is appended to `myname`.

path::replace_filename

Replaces the filename.

```
path& replace_filename(const path& pval);
```

Parameters

pval

The path of the filename.

Remarks

The member function executes:

```
remove_filename();

*this /= pval;
return (*this);
```

path::root_directory

Returns the root directory component of `myname` .

```
path root_directory() const;
```

Remarks

The component may be empty.

path::root_name

Returns the root name component of `myname` .

```
path root_name() const;
```

Remarks

The component may be empty.

path::root_path

Returns the root path component of `myname` .

```
path root_path() const;
```

Remarks

Returns the root path component of `myname` , specifically `root_name()` / `root_directory` . The component may be empty.

path::stem

Returns the `stem` component of `myname` .

```
path stem() const;
```

Remarks

Returns the `stem` component of `myname` , specifically `filename().native()` with any trailing `extension().native()` removed. The component may be empty.

path::string

Converts the sequence stored in `mypath` .

```
template \<class Elem, class Traits = char_traits\<Elem>, class Alloc = allocator\<Elem>>
basic_string\<Elem, Traits, Alloc> string(const Alloc& al = Alloc()) const;
string string() const;
```

Remarks

The first (template) member function converts the sequence stored in `mypath` the same way as:

1. `string()` for `string<char, Traits, Alloc>()`
2. `wstring()` for `string<wchar_t, Traits, Alloc>()`
3. `u16string()` for `string<char16_t, Traits, Alloc>()`
4. `u32string()` for `string<char32_t, Traits, Alloc>()`

The second member function converts the sequence stored in `mypath` to the encoding favored by the host system for a **char** sequence and returns it stored in an object of type `string`.

path::string_type

The type is a synonym for `basic_string<value_type>`.

```
typedef basic_string<value_type> string_type;
```

path::swap

Executes `swap(mypath, right.mypath)`.

```
void swap(path& right) noexcept;
```

path::u16string

Converts the sequence stored in `mypath` to UTF-16 and returns it stored in an object of type `u16string`.

```
u16string u16string() const;
```

path::u32string

Converts the sequence stored in `mypath` to UTF-32 and returns it stored in an object of type `u32string`.

```
u32string u32string() const;
```

path::u8string

Converts the sequence stored in `mypath` to UTF-8 and returns it stored in an object of type `u8string`.

```
string u8string() const;
```

path::value_type

The type describes the `path` elements favored by the host operating system.

```
#if _WIN32_C_LIB
typedef wchar_t value_type;
#else // assume Posix
typedef char value_type;
#endif // filesystem model now defined
```

path::wstring

Converts the sequence stored in `mypath` to the encoding favored by the host system for a **wchar_t** sequence and returns it stored in an object of type `wstring`.

```
wstring wstring() const;
```

See also

[Header Files Reference](#)

recursive_directory_iterator Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an input iterator that sequences through the filenames in a directory, possibly descending into subdirectories recursively. For an iterator `x`, the expression `*x` evaluates to an object of class `directory_entry` that wraps the filename and anything known about its status.

For more information and code examples, see [File System Navigation \(C++\)](#).

Syntax

```
class recursive_directory_iterator;
```

Remarks

The template class stores:

1. an object of type `stack<pair<directory_iterator, path>>`, called `mystack` here for the purposes of exposition, which represents the nest of directories to be sequenced
2. an object of type `directory_entry` called `myentry` here, which represents the current filename in the directory sequence
3. an object of type **bool**, called `no_push` here, which records whether recursive descent into subdirectories is disabled
4. an object of type `directory_options`, called `myoptions` here, which records the options established at construction

A default constructed object of type `recursive_directory_iterator` has an end-of-sequence iterator at `mystack.top().first` and represents the end-of-sequence iterator. For example, given the directory `abc` with entries `def` (a directory), `def/ghi`, and `jkl`, the code:

```
for (recursive_directory_iterator next(path("abc")), end; next != end; ++next)
    visit(next->path());
```

will call `visit` with the arguments `path("abc/def/ghi")` and `path("abc/jkl")`. You can qualify sequencing through a directory subtree in two ways:

1. A directory symlink will be scanned only if you construct a `recursive_directory_iterator` with a `directory_options` argument whose value is `directory_options::follow_directory_symlink`.
2. If you call `disable_recursion_pending` then a subsequent directory encountered during an increment will not be recursively scanned.

Constructors

CONSTRUCTOR	DESCRIPTION
recursive_directory_iterator	Constructs a <code>recursive_directory_iterator</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
depth	Returns <code>mystack.size() - 1</code> , so <code>pval</code> is at depth zero.
disable_recursion_pending	Stores true in <code>no_push</code> .
increment	Advances to the next filename in sequence.
options	Returns <code>myoptions</code> .
pop	Returns the next object.
recursion_pending	Returns <code>!no_push</code> .

Operators

OPERATOR	DESCRIPTION
operator!=	Returns <code>!(*this == right)</code> .
operator=	The defaulted member assignment operators behave as expected.
operator==	Returns true only if both <code>*this</code> and <i>right</i> are end-of-sequence iterators or both are not end-of-sequence-iterators.
operator*	Returns <code>myentry</code> .
operator->	Returns <code>&*this</code> .
operator++	Increments the <code>recursive_directory_iterator</code> .

Requirements

Header: `<filesystem>`

Namespace: `std::tr2::sys`

`recursive_directory_iterator::depth`

Returns `mystack.size() - 1`, so `pval` is at depth zero.

```
int depth() const;
```

`recursive_directory_iterator::disable_recursion_pending`

Stores **true** in `no_push`.

```
void disable_recursion_pending();
```

recursive_directory_iterator::increment

Advances to the next filename in sequence.

```
recursive_directory_iterator& increment(error_code& ec) noexcept;
```

Parameters

ec

Specified error code.

Remarks

The function attempts to advance to the next filename in the nested sequence. If successful, it stores that filename in `myentry`; otherwise it produces an end-of-sequence iterator.

recursive_directory_iterator::operator!=

Returns `!(*this == right)`.

```
bool operator!=(const recursive_directory_iterator& right) const;
```

Parameters

right

The [recursive_directory_iterator](#) for comparison.

recursive_directory_iterator::operator=

The defaulted member assignment operators behave as expected.

```
recursive_directory_iterator& operator=(const recursive_directory_iterator&) = default;  
recursive_directory_iterator& operator=(recursive_directory_iterator&&) noexcept = default;
```

Parameters

recursive_directory_iterator

The [recursive_directory_iterator](#) being copied into the `recursive_directory_iterator`.

recursive_directory_iterator::operator==

Returns **true** only if both `*this` and *right* are end-of-sequence iterators or both are not end-of-sequence iterators.

```
bool operator==(const recursive_directory_iterator& right) const;
```

Parameters

right

The [recursive_directory_iterator](#) for comparison.

recursive_directory_iterator::operator*

Returns `myentry`.

```
const directory_entry& operator*() const;
```

recursive_directory_iterator::operator->

Returns `&*this` .

```
const directory_entry * operator->() const;
```

recursive_directory_iterator::operator++

Increments the `recursive_directory_iterator` .

```
recursive_directory_iterator& operator++();  
recursive_directory_iterator& operator++(int);
```

Parameters

int

The specified increment.

Remarks

The first member function calls `increment()` , then returns `*this` . The second member function makes a copy of the object, calls `increment()` , then returns the copy.

recursive_directory_iterator::options

Returns `myoptions` .

```
directory_options options() const;
```

recursive_directory_iterator::pop

Returns the next object.

```
void pop();
```

Remarks

If `depth() == 0` the object becomes an end-of-sequence iterator. Otherwise, the member function terminates scanning of the current (deepest) directory and resumes at the next lower depth.

recursive_directory_iterator::recursion_pending

Returns `!no_push` .

```
bool recursion_pending() const;
```

recursive_directory_iterator::recursive_directory_iterator

Constructs a `recursive_directory_iterator`.

```
recursive_directory_iterator() noexcept;
explicit recursive_directory_iterator(const path& pval);

recursive_directory_iterator(const path& pval,
    error_code& ec) noexcept;
recursive_directory_iterator(const path& pval,
    directory_options opts);

recursive_directory_iterator(const path& pval,
    directory_options opts,
    error_code& ec) noexcept;
recursive_directory_iterator(const recursive_directory_iterator&) = default;
recursive_directory_iterator(recursive_directory_iterator&&) noexcept = default;
```

Parameters

pval

The specified path.

error_code

The specified error code.

opts

The specified directory options.

recursive_directory_iterator

The `recursive_directory_iterator` of which the constructed `recursive_directory_iterator` is to be a copy.

Remarks

The first constructor produces an end-of-sequence iterator. The second and third constructors store **false** in `no_push` and `directory_options::none` in `myoptions`, then attempt to open and read *pval* as a directory. If successful, they initialize `mystack` and `myentry` to designate the first non-directory filename in the nested sequence; otherwise they produce an end-of-sequence iterator.

The fourth and fifth constructors behave the same as the second and third, except that they first store *opts* in `myoptions`. The defaulted constructors behave as expected.

See also

[Header Files Reference](#)

[<filesystem>](#)

[File System Navigation \(C++\)](#)

space_info Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Holds information about a volume.

Syntax

```
struct space_info
{
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};
```

Members

Public Data Members

NAME	DESCRIPTION
<code>unsigned long long capacity</code>	Represents the total number of bytes that the volume can represent.
<code>unsigned long long free</code>	Represents the number of bytes that are not used to represent data on the volume.
<code>unsigned long long available</code>	Represents the number of bytes that are available to represent data on the volume.

Requirements

Header: <filesystem>

Namespace: std::experimental::filesystem

See also

[Header Files Reference](#)

[<filesystem>](#)

[File System Navigation \(C++\)](#)

<forward_list>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class `forward_list` and several supporting templates.

Syntax

```
#include <forward_list>
```

Operators

OPERATOR	DESCRIPTION
<code>operator==</code>	Tests if the forward list object on the left side of the operator is equal to the forward list object on the right side.
<code>operator!=</code>	Tests if the forward list object on the left side of the operator is not equal to the forward list object on the right side.
<code>operator<</code>	Tests if the forward list object on the left side of the operator is less than the forward list object on the right side.
<code>operator<=</code>	Tests if the forward list object on the left side of the operator is less than or equal to the forward list object on the right side.
<code>operator></code>	Tests if the forward list object on the left side of the operator is greater than the forward list object on the right side.
<code>operator>=</code>	Tests if the forward list object on the left side of the operator is greater than or equal to the forward list object on the right side.

Functions

FUNCTION	DESCRIPTION
<code>swap</code>	Exchanges the elements of two forward lists.

Classes

CLASS	DESCRIPTION
<code>forward_list</code>	Describes an object that controls a varying-length sequence of elements. The sequence is stored as a singly-linked list of elements, each containing a member of type <code>Type</code> .

See also

[Header Files Reference](#)

<forward_list> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

swap

Exchanges the elements of two forward lists.

```
void swap(  
    forward_list <Type, Allocator>& left,  
    forward_list <Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Remarks

This template function executes `left.swap(right)` .

See also

[<forward_list>](#)

<forward_list> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator==

Tests if the forward list object on the left side of the operator is equal to the forward list object on the right side.

```
bool operator==(
    const forward_list<Type, Allocator>& left,
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Remarks

This template function overloads `operator==` to compare two objects of template class `forward_list`. The function returns

```
distance(left.begin(), end()) == distance(right.begin(),right.end()) && equal(left.begin(),left.end(),right.begin())
```

operator!=

Tests if the forward list object on the left side of the operator is not equal to the forward list object on the right side.

```
bool operator!=(
    const forward_list<Type, Allocator>& left,
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Return Value

true if the lists are not equal; **false** if the lists are equal.

Remarks

This template function returns `!(left == right)`.

operator<

Tests if the forward list object on the left side of the operator is less than the forward list object on the right side.

```
bool operator<(
    const forward_list<Type, Allocator>& left,
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Return Value

true if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise **false**.

Remarks

This template function overloads `operator<` to compare two objects of template class `forward_list`. The function returns `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

Tests if the forward list object on the left side of the operator is less than or equal to the forward list object on the right side.

```
bool operator<=(
    const forward_list<Type, Allocator>& left,
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Return Value

true if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise **false**.

Remarks

This template function returns `!(right < left)`.

operator>

Tests if the forward list object on the left side of the operator is greater than the forward list object on the right side.

```
bool operator>(  
    const forward_list<Type, Allocator>& left,  
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Return Value

true if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **false**.

Remarks

This template function returns `right < left` .

operator>=

Tests if the forward list object on the left side of the operator is greater than or equal to the forward list object on the right side.

```
bool operator>=(  
    const forward_list<Type, Allocator>& left,  
    const forward_list<Type, Allocator>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>forward_list</code> .
<i>right</i>	An object of type <code>forward_list</code> .

Return Value

true if the forward list on the left side of the operator is greater than or equal to the forward list on the right side of the operator; otherwise **false**.

Remarks

The template function returns `!(left < right)` .

See also

[<forward_list>](#)

forward_list Class

4/29/2019 • 21 minutes to read • [Edit Online](#)

Describes an object that controls a varying-length sequence of elements. The sequence is stored as a singly-linked list of nodes, each containing a member of type `Type`.

Syntax

```
template <class Type,  
         class Allocator = allocator<Type>>  
class forward_list
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The element data type to be stored in the <code>forward_list</code> .
<i>Allocator</i>	The stored allocator object that encapsulates details about the <code>forward_list</code> allocation and deallocation of memory. This parameter is optional. The default value is <code>allocator<Type></code> .

Remarks

A `forward_list` object allocates and frees storage for the sequence it controls through a stored object of class *Allocator* that is based on [allocator Class](#) (commonly known as `std::allocator`). For more information, see [Allocators](#). An allocator object must have the same external interface as an object of template class `allocator`.

NOTE

The stored allocator object is not copied when the container object is assigned.

Iterators, pointers and references might become invalid when elements of their controlled sequence are erased through `forward_list`. Insertions and splices performed on the controlled sequence through `forward_list` do not invalidate iterators.

Additions to the controlled sequence might occur by calls to `forward_list::insert_after`, which is the only member function that calls the constructor `Type(const T&)`. `forward_list` might also call move constructors. If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of template class `forward_list` is left in a known state when such exceptions occur.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>forward_list</code>	Constructs an object of type <code>forward_list</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the allocator class for a forward list object.
<code>const_iterator</code>	A type that provides a constant iterator for the forward list.
<code>const_pointer</code>	A type that provides a pointer to a const element in a forward list.
<code>const_reference</code>	A type that provides a constant reference to an element in the forward list.
<code>difference_type</code>	A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.
<code>iterator</code>	A type that provides an iterator for the forward list.
<code>pointer</code>	A type that provides a pointer to an element in the forward list.
<code>reference</code>	A type that provides a reference to an element in the forward list.
<code>size_type</code>	A type that represents the unsigned distance between two elements.
<code>value_type</code>	A type that represents the type of element stored in a forward list.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Erases elements from a forward list and copies a new set of elements to a target forward list.
<code>before_begin</code>	Returns an iterator addressing the position before the first element in a forward list.
<code>begin</code>	Returns an iterator addressing the first element in a forward list.
<code>cbegin</code>	Returns a const iterator addressing the first element in a forward list.
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a forward list.
<code>clear</code>	Erases all the elements of a forward list.

MEMBER FUNCTION	DESCRIPTION
<code>emplace_after</code>	Move constructs a new element after a specified position.
<code>emplace_front</code>	Adds an element constructed in place to the beginning of the list.
<code>empty</code>	Tests whether a forward list is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a forward list.
<code>erase_after</code>	Removes elements from the forward list after a specified position.
<code>front</code>	Returns a reference to the first element in a forward list.
<code>get_allocator</code>	Returns a copy of the allocator object used to construct a forward list.
<code>insert_after</code>	Adds elements to the forward list after a specified position.
<code>max_size</code>	Returns the maximum length of a forward list.
<code>merge</code>	Removes the elements from the argument list, inserts them into the target forward list, and orders the new, combined set of elements in ascending order or in some other specified order.
<code>pop_front</code>	Deletes the element at the beginning of a forward list.
<code>push_front</code>	Adds an element to the beginning of a forward list.
<code>remove</code>	Erases elements in a forward list that matches a specified value.
<code>remove_if</code>	Erases elements from a forward list for which a specified predicate is satisfied.
<code>resize</code>	Specifies a new size for a forward list.
<code>reverse</code>	Reverses the order in which the elements occur in a forward list.
<code>sort</code>	Arranges the elements in ascending order or with an order specified by a predicate.
<code>splice_after</code>	Restitches links between nodes.
<code>swap</code>	Exchanges the elements of two forward lists.
<code>unique</code>	Removes adjacent elements that pass a specified test.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Replaces the elements of the forward list with a copy of another forward list.

Requirements

Header: `<forward_list>`

Namespace: `std`

`forward_list::allocator_type`

A type that represents the allocator class for a forward list object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

`forward_list::assign`

Erases elements from a forward list and copies a new set of elements to a target forward list.

```
void assign(
    size_type Count,
    const Type& Val);

void assign(
    initializer_list<Type> IList);

template <class InputIterator>
void assign(InputIterator First, InputIterator Last);
```

Parameters

PARAMETER	DESCRIPTION
<i>first</i>	The beginning of the replacement range.
<i>last</i>	The end of the replacement range.
<i>count</i>	The number of elements to assign.
<i>val</i>	The value to assign each element.
<i>Type</i>	The type of the value.
<i>IList</i>	The <code>initializer_list</code> to copy.

Remarks

If the `forward_list` is an integer type, the first member function behaves the same as

`assign((size_type)First, (Type)Last)`. Otherwise, the first member function replaces the sequence controlled by

`*this` with the sequence [`First`, `Last`), which must not overlap the initial controlled sequence.

The second member function replaces the sequence controlled by `*this` with a repetition of `Count` elements of value `Val`.

The third member function copies the elements of the `initializer_list` into the `forward_list`.

`forward_list::before_begin`

Returns an iterator addressing the position before the first element in a forward list.

```
const_iterator before_begin() const;
iterator before_begin();
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

Remarks

`forward_list::begin`

Returns an iterator addressing the first element in a forward list.

```
const_iterator begin() const;
iterator begin();
```

Return Value

A forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Remarks

`forward_list::cbefore_begin`

Returns a const iterator addressing the position before the first element in a forward list.

```
const_iterator cbefore_begin() const;
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

Remarks

`forward_list::cbegin`

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();
// i2 is Container<T>::const_iterator
```

forward_list::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

forward_list::clear

Erases all the elements of a forward list.

```
void clear();
```

Remarks

This member function calls `erase_after(before_begin(), end())`.

forward_list::const_iterator

A type that provides a constant iterator for the forward list.

```
typedef implementation-defined const_iterator;
```

Remarks

`const_iterator` describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for an implementation-defined type.

forward_list::const_pointer

A type that provides a pointer to a **const** element in a forward list.

```
typedef typename Allocator::const_pointer  
const_pointer;
```

Remarks

forward_list::const_reference

A type that provides a constant reference to an element in the forward list.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

forward_list::difference_type

A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

`difference_type` describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

forward_list::emplace_after

Move constructs a new element after a specified position.

```
template <class T>  
iterator emplace_after(const_iterator Where, Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>Where</i>	The position in the target forward list where the new element is constructed.
<i>val</i>	The constructor argument.

Return Value

An iterator that designates the newly inserted element.

Remarks

This member function inserts an element with the constructor arguments *val* just after the element pointed to by *Where* in the controlled sequence. Its behavior is otherwise the same as [forward_list::insert_after](#).

forward_list::emplace_front

Adds an element constructed in place to the beginning of the list.

```
template <class Type>
void emplace_front(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the forward list.

Remarks

This member function inserts an element with the constructor arguments `_ val` at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

forward_list::empty

Tests whether a forward list is empty.

```
bool empty() const;
```

Return Value

true if the forward list is empty; otherwise, **false**.

forward_list::end

Returns an iterator that addresses the location succeeding the last element in a forward list.

```
const_iterator end() const;
iterator end();
```

Return Value

A forward iterator that points just beyond the end of the sequence.

forward_list::erase_after

Removes elements from the forward list after a specified position.

```
iterator erase_after(const_iterator Where);
iterator erase_after(const_iterator first, const_iterator last);
```

Parameters

PARAMETER	DESCRIPTION
<i>Where</i>	The position in the target forward list where the element is erased.
<i>first</i>	The beginning of the range to erase.
<i>last</i>	The end of the range to erase.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or `forward_list::end` if no such element exists.

Remarks

The first member function removes the element of the controlled sequence just after *Where*.

The second member function removes the elements of the controlled sequence in the range `(first, last)` (neither end point is included).

Erasing `N` elements causes `N` destructor calls. [Reallocation](#) occurs, so iterators and references become invalid for the erased elements.

The member functions never throw an exception.

forward_list::forward_list

Constructs an object of type `forward_list`.

```
forward_list();
explicit forward_list(const Allocator& Al);
explicit forward_list(size_type Count);
forward_list(size_type Count, const Type& Val);
forward_list(size_type Count, const Type& Val, const Allocator& Al);
forward_list(const forward_list& Right);
forward_list(const forward_list& Right, const Allocator& Al);
forward_list(forward_list&& Right);
forward_list(forward_list&& Right, const Allocator& Al);
forward_list(initializer_list<Type> IList, const Alloc& Al);
template <class InputIterator>
forward_list(InputIterator First, InputIterator Last);
template <class InputIterator>
forward_list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The allocator class to use with this object.
<i>Count</i>	The number of elements in the list constructed.
<i>Val</i>	The value of the elements in the list constructed.
<i>Right</i>	The list of which the constructed list is to be a copy.

PARAMETER	DESCRIPTION
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.
<i>lList</i>	The initializer_list to copy.

Remarks

All constructors store an [allocator](#) and initialize the controlled sequence. The allocator object is the argument *Al*, if present. For the copy constructor, it is `right.get_allocator()`. Otherwise, it is `Allocator()`.

The first two constructors specify an empty initial controlled sequence.

The third constructor specifies a repetition of *Count* elements of value `Type()`.

The fourth and fifth constructors specify a repetition of *Count* elements of value *Val*.

The sixth constructor specifies a copy of the sequence controlled by *Right*. If `InputIterator` is an integer type, the next two constructors specify a repetition of `(size_type)First` elements of value `(Type)Last`. Otherwise, the next two constructors specify the sequence `[First, Last)`.

The ninth and tenth constructors are the same as the sixth, but with an [rvalue](#) reference.

The last constructor specifies the initial controlled sequence with an object of class `initializer_list<Type>`.

forward_list::front

Returns a reference to the first element in a forward list.

```
reference front();
const_reference front() const;
```

Return Value

A reference to the first element of the controlled sequence, which must be non-empty.

forward_list::get_allocator

Returns a copy of the allocator object used to construct a forward list.

```
allocator_type get_allocator() const;
```

Return Value

The stored [allocator](#) object.

forward_list::insert_after

Adds elements to the forward list after a specified position.


```

iterator insert_after(const_iterator Where, const Type& Val);
void insert_after(const_iterator Where, size_type Count, const Type& Val);
void insert_after(const_iterator Where, initializer_list<Type> Ilist);
iterator insert_after(const_iterator Where, Type&& Val);
template <class InputIterator>
void insert_after(const_iterator Where, InputIterator First, InputIterator Last);

```

Parameters

PARAMETER	DESCRIPTION
<i>Where</i>	The position in the target forward list where the first element is inserted.
<i>Count</i>	The number of elements to insert.
<i>First</i>	The beginning of the insertion range.
<i>Last</i>	The end of the insertion range.
<i>Val</i>	The element added to the forward list.
<i>Ilist</i>	The <code>initializer_list</code> to insert.

Return Value

An iterator that designates the newly inserted element (first and last member functions only).

Remarks

Each of the member functions inserts—just after the element pointed to by *Where* in the controlled sequence—a sequence that's specified by the remaining operands.

The first member function inserts an element that has value *Val* and returns an iterator that designates the newly inserted element.

The second member function inserts a repetition of *Count* elements of value *Val*.

If `InputIterator` is an integer type, the third member function behaves the same as `insert(it, (size_type)First, (Type)Last)`. Otherwise, it inserts the sequence `[First, Last)`, which must not overlap the initial controlled sequence.

The fourth member function inserts the sequence that's specified by an object of class `initializer_list<Type>`.

The last member function is the same as the first, but with an [rvalue](#) reference.

Inserting `N` elements causes `N` constructor calls. [Reallocation](#) occurs, but no iterators or references become invalid.

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

forward_list::iterator

A type that provides an iterator for the forward list.

```
typedef implementation-defined iterator;
```

Remarks

`iterator` describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for an implementation-defined type.

forward_list::max_size

Returns the maximum length of a forward list.

```
size_type max_size() const;
```

Return Value

The length of the longest sequence that the object can control.

Remarks

forward_list::merge

Combines two sorted sequences into a single sorted sequence in linear time. Removes the elements from the argument list, and inserts them into this `forward_list`. The two lists should be sorted by the same compare function object before the call to `merge`. The combined list will be sorted by that compare function object.

```
void merge(forward_list& right);  
template <class Predicate>  
void merge(forward_list& right, Predicate comp);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The forward list to merge from.
<i>comp</i>	The compare function object that is used to sort elements.

Remarks

`forward_list::merge` removes the elements from the `forward_list` `right`, and inserts them into this `forward_list`. Both sequences must be ordered by the same predicate, described below. The combined sequence is also ordered by that compare function object.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The second member function imposes the order `! comp(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `right`.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

forward_list::operator=

Replaces the elements of the forward list with a copy of another forward list.

```
forward_list& operator=(const forward_list& right);
forward_list& operator=(initializer_list<Type> IList);
forward_list& operator=(forward_list&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The forward list being copied into the forward list.
<i>IList</i>	A brace-enclosed initializer list, which behaves just like a sequence of elements of type <code>Type</code> .

Remarks

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *right*.

The second member operator replaces the controlled sequence from an object of class `initializer_list<Type>` .

The third member operator is the same as the first, but with an [rvalue](#) reference.

forward_list::pointer

A type that provides a pointer to an element in the forward list.

```
typedef typename Allocator::pointer pointer;
```

Remarks

forward_list::pop_front

Deletes the element at the beginning of a forward list.

```
void pop_front();
```

Remarks

The first element of the forward list must be non-empty.

The member function never throws an exception.

forward_list::push_front

Adds an element to the beginning of a forward list.

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the forward list.

Remarks

If an exception is thrown, the container is left unaltered and the exception is rethrown.

forward_list::reference

A type that provides a reference to an element in the forward list.

```
typedef typename Allocator::reference reference;
```

Remarks

forward_list::remove

Erases elements in a forward list that matches a specified value.

```
void remove(const Type& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value which, if held by an element, will result in that element's removal from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == val`.

The member function never throws an exception.

forward_list::remove_if

Erases elements from a forward list for which a specified predicate is satisfied.

```
template <class Predicate>
void remove_if(Predicate pred);
```

Parameters

PARAMETER	DESCRIPTION
<i>pred</i>	The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `pred(*P)` is true.

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

forward_list::resize

Specifies a new size for a forward list.

```
void resize(size_type _Newsize);
void resize(size_type _Newsize, const Type& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>_Newsize</i>	The number of elements in the resized forward list.
<i>val</i>	The value to use for padding.

Remarks

The member functions both ensure that the number of elements in the list henceforth is *_Newsize*. If it must make the controlled sequence longer, the first member function appends elements with value `Type()`, while the second member function appends elements with value *val*. To make the controlled sequence shorter, both member functions effectively call `erase_after(begin() + _Newsize - 1, end())`.

forward_list::reverse

Reverses the order in which the elements occur in a forward list.

```
void reverse();
```

Remarks

forward_list::size_type

A type that represents the unsigned distance between two elements.

```
typedef typename Allocator::size_type size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence.

forward_list::sort

Arranges the elements in ascending order or with an order specified by a predicate.

```
void sort();
template <class Predicate>
void sort(Predicate pred);
```

Parameters

PARAMETER	DESCRIPTION
<i>pred</i>	The ordering predicate.

Remarks

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The member template function imposes the order `! pred(*Pj, *Pi)` whenever `i < j`. No ordered pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. (The sort is stable.)

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

forward_list::splice_after

Removes elements from a source `forward_list` and inserts them into a destination `forward_list`.

```
// insert the entire source forward_list
void splice_after(const_iterator Where, forward_list& Source);
void splice_after(const_iterator Where, forward_list&& Source);

// insert one element of the source forward_list
void splice_after(const_iterator Where, forward_list& Source, const_iterator Iter);
void splice_after(const_iterator Where, forward_list&& Source, const_iterator Iter);

// insert a range of elements from the source forward_list
void splice_after(
    const_iterator Where,
    forward_list& Source,
    const_iterator First,
    const_iterator Last);

void splice_after(
    const_iterator Where,
    forward_list&& Source,
    const_iterator First,
    const_iterator Last);
```

Parameters

Where

The position in the destination `forward_list` after which to insert.

Source

The source `forward_list` that is to be inserted into the destination `forward_list`.

Iter

The element to be inserted from the source `forward_list`.

First

The first element in the range to be inserted from source `forward_list`.

Last

The first position beyond the range to be inserted from the source `forward_list`.

Remarks

The first pair of member functions inserts the sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes all elements from *Source*. (`&Source` must not equal **this**.)

The second pair of member functions removes the element just after *Iter* in the sequence controlled by *Source* and inserts it just after the element in the controlled sequence pointed to by *Where*. (If

```
Where == Iter || Where == ++Iter
```

, no change occurs.)

The third pair of member functions (ranged splice) inserts the subrange designated by `(First, Last)` from the

sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes the original subrange from the sequence controlled by *Source*. (If `&Source == this`, the range `(First, Last)` must not include the element pointed to by *Where*.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class `iterator` is incremented `N` times.

No iterators, pointers, or references that designate spliced elements become invalid.

Example

```

// forward_list_splice_after.cpp
// compile with: /EHsc /W4
#include <forward_list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    forward_list<int> c1{ 10, 11 };
    forward_list<int> c2{ 20, 21, 22 };
    forward_list<int> c3{ 30, 31 };
    forward_list<int> c4{ 40, 41, 42, 43 };

    forward_list<int>::iterator where_iter;
    forward_list<int>::iterator first_iter;
    forward_list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice_after(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);

    first_iter = c3.begin();
    c2.splice_after(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);

    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    c2.splice_after(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}

```



```
Beginning state of lists:c1 = (10) (11)c2 = (20) (21) (22)c3 = (30) (31)c4 = (40) (41) (42) (43)After splicing
c1 into c2:c1 =c2 = (20) (21) (10) (11) (22)After splicing the first element of c3 into c2:c3 = (30)c2 = (20)
(21) (31) (10) (11) (22)After splicing a range of c4 into c2:c4 = (40) (41)c2 = (20) (21) (42) (43) (31) (10)
(11) (22)
```

forward_list::swap

Exchanges the elements of two forward lists.

```
void swap(forward_list& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The forward list providing the elements to be exchanged.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

forward_list::unique

Eliminates all but the first element from every consecutive group of equal elements.

```
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate comp);
```

Parameters

PARAMETER	DESCRIPTION
<i>comp</i>	The binary predicate used to compare successive elements.

Remarks

Keeps the first of each unique element, and removes the rest. The elements must be sorted so that elements of equal value are adjacent in the list.

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the second member function removes every element for which `i + 1 == j && comp(*Pi, *Pj)`.

For a controlled sequence of length `N` (> 0), the predicate `comp(*Pi, *Pj)` is evaluated `N - 1` times.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

forward_list::value_type

A type that represents the type of element stored in a forward list.

```
typedef typename Allocator::value_type value_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

See also

[<forward_list>](#)

<fstream>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines several classes that support iostreams operations on sequences stored in external files.

Syntax

```
#include <fstream>
```

Typedefs

TYPE NAME	DESCRIPTION
filebuf	A type <code>basic_filebuf</code> specialized on char template parameters.
fstream	A type <code>basic_fstream</code> specialized on char template parameters.
ifstream	A type <code>basic_ifstream</code> specialized on char template parameters.
ofstream	A type <code>basic_ofstream</code> specialized on char template parameters.
wfstream	A type <code>basic_fstream</code> specialized on wchar_t template parameters.
wifstream	A type <code>basic_ifstream</code> specialized on wchar_t template parameters.
wofstream	A type <code>basic_ofstream</code> specialized on wchar_t template parameters.
wfilebuf	A type <code>basic_filebuf</code> specialized on wchar_t template parameters.

Classes

CLASS	DESCRIPTION
basic_filebuf	The template class describes a stream buffer that controls the transmission of elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> , to and from a sequence of elements stored in an external file.

CLASS	DESCRIPTION
basic_fstream	The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_filebuf < Elem , Tr >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> .
basic_ifstream	The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_filebuf < Elem , Tr >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> .
basic_ofstream	The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_filebuf < Elem , Tr >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> .

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<fstream> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

filebuf	fstream	ifstream
ofstream	wfilebuf	wfstream
wifstream	wofstream	

filebuf

A type `basic_filebuf` specialized on **char** template parameters.

```
typedef basic_filebuf<char, char_traits<char>> filebuf;
```

Remarks

The type is a synonym for template class [basic_filebuf](#), specialized for elements of type **char** with default character traits.

fstream

A type `basic_fstream` specialized on **char** template parameters.

```
typedef basic_fstream<char, char_traits<char>> fstream;
```

Remarks

The type is a synonym for template class [basic_fstream](#), specialized for elements of type **char** with default character traits.

ifstream

Defines a stream to be used to read single-byte character data serially from a file. `ifstream` is a typedef that specializes the template class `basic_ifstream` for **char**.

There is also `wifstream`, a typedef that specializes `basic_ifstream` to read **wchar_t** double-wide characters. For more information, see [wifstream](#).

```
typedef basic_ifstream<char, char_traits<char>> ifstream;
```

Remarks

The type is a synonym for template class [basic_ifstream](#), specialized for elements of type char with default character traits. An example is

```
using namespace std;

ifstream infile("existingtextfile.txt");

if (!infile.bad())
{
    // Dump the contents of the file to cout.
    cout << infile.rdbuf();infile.close();
}
```

ofstream

A type `basic_ofstream` specialized on **char** template parameters.

```
typedef basic_ofstream<char, char_traits<char>> ofstream;
```

Remarks

The type is a synonym for template class [basic_ofstream](#), specialized for elements of type **char** with default character traits.

wfstream

A type `basic_fstream` specialized on **wchar_t** template parameters.

```
typedef basic_fstream<wchar_t, char_traits<wchar_t>> wfstream;
```

Remarks

The type is a synonym for template class [basic_fstream](#), specialized for elements of type **wchar_t** with default character traits.

wifstream

A type `basic_ifstream` specialized on **wchar_t** template parameters.

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t>> wifstream;
```

Remarks

The type is a synonym for template class [basic_ifstream](#), specialized for elements of type **wchar_t** with default character traits.

wofstream

A type `basic_ofstream` specialized on **wchar_t** template parameters.

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t>> wofstream;
```

Remarks

The type is a synonym for template class [basic_ofstream](#), specialized for elements of type **wchar_t** with default character traits.

wfilebuf

A type `basic_filebuf` specialized on **wchar_t** template parameters.

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t>> wfilebuf;
```

Remarks

The type is a synonym for template class [basic_filebuf](#), specialized for elements of type **wchar_t** with default character traits.

See also

[<fstream>](#)

basic_filebuf Class

11/8/2018 • 15 minutes to read • [Edit Online](#)

Describes a stream buffer that controls the transmission of elements of type *Elem*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_filebuf : public basic_streambuf<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually `char_traits < Elem >`).

Remarks

The template class describes a stream buffer that controls the transmission of elements of type *Elem*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

NOTE

Objects of type `basic_filebuf` are created with an internal buffer of type `char *` regardless of the `char_type` specified by the type parameter *Elem*. This means that a Unicode string (containing `wchar_t` characters) will be converted to an ANSI string (containing `char` characters) before it is written to the internal buffer. To store Unicode strings in the buffer, create a new buffer of type `wchar_t` and set it using the `basic_streambuf::pubsetbuf()` method. To see an example that demonstrates this behavior, see below.

An object of class `basic_filebuf < Elem, Tr >` stores a file pointer, which designates the `FILE` object that controls the stream associated with an open file. It also stores pointers to two file conversion facets for use by the protected member functions `overflow` and `underflow`. For more information, see `basic_filebuf::open`.

Example

The following example demonstrates how to force an object of type `basic_filebuf<wchar_t>` to store Unicode characters in its internal buffer by calling the `pubsetbuf()` method.

```
// unicode_basic_filebuf.cpp
// compile with: /EHsc

#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include <memory.h>
#include <string.h>

#define IBUFSIZE 16
```



```

using namespace std;

void hexdump(const string& filename);

int main()
{
    wchar_t* wszHello = L"Hello World";
    wchar_t wBuffer[128];

    basic_filebuf<wchar_t> wOutFile;

    // Open a file, wcHello.txt, then write to it, then dump the
    // file's contents in hex
    wOutFile.open("wcHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wcHello.txt\n";
        return -1;
    }
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "Hex Dump of wcHello.txt - note that output is ANSI chars:\n";
    hexdump(string("wcHello.txt"));

    // Open a file, wwHello.txt, then set the internal buffer of
    // the basic_filebuf object to be of type wchar_t, then write
    // to the file and dump the file's contents in hex
    wOutFile.open("wwHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wwHello.txt\n";
        return -1;
    }
    wOutFile.pubsetbuf(wBuffer, (streamsize)128);
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "\nHex Dump of wwHello.txt - note that output is wchar_t chars:\n";
    hexdump(string("wwHello.txt"));

    return 0;
}

// dump contents of filename to stdout in hex
void hexdump(const string& filename)
{
    ifstream ifile(filename.c_str(),
        ios_base::in | ios_base::binary);
    char *ibuff = new char[IBUFSIZE];
    char *obuff = new char[(IBUFSIZE*2)+1];
    int i;

    if(!ifile.is_open())
    {
        cout << "Cannot Open " << filename.c_str()
            << " for reading\n";
        return;
    }
    if(!ibuff || !obuff)
    {
        cout << "Cannot Allocate buffers\n";
        ifile.close();
        return;
    }

    while(!ifile.eof())
    {

```

```

memset(obuff,0,(IBUFSIZE*2)+1);
memset(ibuff,0,IBUFSIZE);
ifile.read(ibuff,IBUFSIZE);

// corner case where file is exactly a multiple of
// 16 bytes in length
if(ibuff[0] == 0 && ifile.eof())
    break;

for(i = 0; i < IBUFSIZE; i++)
{
    if(ibuff[i] >= ' ')
        obuff[i] = ibuff[i];
    else
        obuff[i] = '.';

    cout << setfill('0') << setw(2) << hex
        << (int)ibuff[i] << ' ';
}
cout << " " << obuff << endl;
}
ifile.close();
}

```

Hex Dump of wcHello.txt - note that output is ANSI chars:

```
48 65 6c 6c 6f 20 57 6f 72 6c 64 00 00 00 00 00 Hello World.....
```

Hex Dump of wwHello.txt - note that output is wchar_t chars:

```
48 00 65 00 6c 00 6c 00 6f 00 20 00 57 00 6f 00 H.e.l.l.o. .W.o.
72 00 6c 00 64 00 00 00 00 00 00 00 00 00 00 00 r.l.d.....
```

Constructors

CONSTRUCTOR	DESCRIPTION
basic_filebuf	Constructs an object of type <code>basic_filebuf</code> .

Typedefs

TYPE NAME	DESCRIPTION
char_type	Associates a type name with the <code>Elem</code> template parameter.
int_type	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <code>Tr</code> scope.
off_type	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <code>Tr</code> scope.
pos_type	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <code>Tr</code> scope.
traits_type	Associates a type name with the <code>Tr</code> template parameter.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>close</code>	Closes a file.
<code>is_open</code>	Indicates whether a file is open.
<code>open</code>	Opens a file.
<code>overflow</code>	A protected virtual function that can be called when a new character is inserted into a full buffer.
<code>pbackfail</code>	The protected virtual member function tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).
<code>seekoff</code>	The protected virtual member function tries to alter the current positions for the controlled streams.
<code>seekpos</code>	The protected virtual member function tries to alter the current positions for the controlled streams.
<code>setbuf</code>	The protected virtual member function performs an operation particular to each derived stream buffer.
<code>Swap</code>	Exchanges the content of this <code>basic_filebuf</code> for the content of the provided <code>basic_filebuf</code> parameter.
<code>sync</code>	Protected, virtual function tries to synchronize the controlled streams with any associated external streams.
<code>uflow</code>	Protected, virtual function to extract the current element from the input stream.
<code>underflow</code>	Protected, virtual function to extract the current element from the input stream.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_filebuf::basic_filebuf`

Constructs an object of type `basic_filebuf`.

```
basic_filebuf();

basic_filebuf(basic_filebuf&& right);
```

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores a null pointer in the file pointer.

The second constructor initializes the object with the contents of `right`, treated as an rvalue reference.

basic_filebuf::char_type

Associates a type name with the `Elem` template parameter.

```
typedef Elem char_type;
```

basic_filebuf::close

Closes a file.

```
basic_filebuf<Elem, Tr> *close();
```

Return Value

The member function returns a null pointer if the file pointer is a null pointer.

Remarks

`close` calls `fclose` (**fp**). If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **this** to indicate that the file was successfully closed.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls `overflow`. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet `fac` to call `fac.unshift` as needed. Each element `byte` of type **char** thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc` (**byte, fp**). If the call to `fac.unshift` or any write fails, the function does not succeed.

Example

The following sample assumes two files in the current directory: `basic_filebuf_close.txt` (contents is "testing") and `iotest.txt` (contents is "ssss").

```
// basic_filebuf_close.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main() {
    using namespace std;
    ifstream file;
    basic_ifstream <wchar_t> wfile;
    char c;
    // Open and close with a basic_filebuf
    file.rdbuf()->open( "basic_filebuf_close.txt", ios::in );
    file >> c;
    cout << c << endl;
    file.rdbuf( )->close( );

    // Open/close directly
    file.open( "iotest.txt" );
    file >> c;
    cout << c << endl;
    file.close( );

    // open a file with a wide character name
    wfile.open( L"iotest.txt" );

    // Open and close a nonexistent with a basic_filebuf
    file.rdbuf()->open( "ziotest.txt", ios::in );
    cout << file.fail() << endl;
    file.rdbuf( )->close( );

    // Open/close directly
    file.open( "ziotest.txt" );
    cout << file.fail() << endl;
    file.close( );
}
```

```
t
s
0
1
```

basic_filebuf::int_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::int_type int_type;
```

basic_filebuf::is_open

Indicates whether a file is open.

```
bool is_open() const;
```

Return Value

true if the file pointer is not a null pointer.

Example

```
// basic_filebuf_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
    using namespace std;
    ifstream file;
    cout << boolalpha << file.rdbuf( )->is_open( ) << endl;

    file.open( "basic_filebuf_is_open.cpp" );
    cout << file.rdbuf( )->is_open( ) << endl;
}
```

```
false
true
```

basic_filebuf::off_type

Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::off_type off_type;
```

basic_filebuf::open

Opens a file.

```
basic_filebuf<Elem, Tr> *open(
    const char* _Filename,
    ios_base::openmode _Mode,
    int _Prot = (int)ios_base::_Openprot);

basic_filebuf<Elem, Tr> *open(
    const char* _Filename,
    ios_base::openmode _Mode);

basic_filebuf<Elem, Tr> *open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode,
    int _Prot = (int)ios_base::_Openprot);

basic_filebuf<Elem, Tr> *open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the *shflag* parameter in [_fsopen](#), [_wfsopen](#).

Return Value

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns **this**.

Remarks

The member function opens the file with filename *filename*, by calling `fopen(filename, strmode)`. `strmode` is determined from **mode** & ~(`ate` & | `binary`):

- `ios_base::in` becomes "r" (open existing file for reading).
- `ios_base::out` or `ios_base::out | ios_base::trunc` becomes "w" (truncate existing file or create for writing).
- `ios_base::out | app` becomes "a" (open existing file for appending all writes).
- `ios_base::in | ios_base::out` becomes "r+" (open existing file for reading and writing).
- `ios_base::in | ios_base::out | ios_base::trunc` becomes "w+" (truncate existing file or create for reading and writing).
- `ios_base::in | ios_base::out | ios_base::app` becomes "a+" (open existing file for reading and for appending all writes).

If **mode** & `ios_base::binary` is nonzero, the function appends `b` to `strmode` to open a binary stream instead of a text stream. It then stores the value returned by `fopen` in the file pointer `fp`. If **mode** & `ios_base::ate` is nonzero and the file pointer is not a null pointer, the function calls `fseek(fp, 0, SEEK_END)` to position the stream at end of file. If that positioning operation fails, the function calls `close(fp)` and stores a null pointer in the file pointer.

If the file pointer is not a null pointer, the function determines the file conversion facet: `use_facet < codecvt < Elem, char, traits_type::state_type> >(getloc)`, for use by `underflow` and `overflow`.

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns **this**.

Example

See `basic_filebuf::close` for an example that uses `open`.

basic_filebuf::operator=

Assign the content of this stream buffer object. This is a move assignment involving an rvalue that does not leave a copy behind.

```
basic_filebuf& operator=(basic_filebuf&& right);
```

Parameters

right

An rvalue reference to a `basic_filebuf` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *right*, treated as an rvalue reference. For more information, see [Rvalue Reference Declarator: &&](#).

basic_filebuf::overflow

Called when a new character is inserted into a full buffer.

```
virtual int_type overflow(int_type _Meta = traits_type::eof);
```

Parameters

_Meta

The character to insert into the buffer or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof`. Otherwise, it returns **`traits_type::not_eof`**(*_Meta*).

Remarks

If *_Meta* != **`traits_type::eof`**, the protected virtual member function endeavors to insert the element **`ch = traits_type::to_char_type`**(*_Meta*) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by `ch`, by using the file conversion facet `fac` to call `fac.out` as needed. Each element `ch` of type *char* thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(ch, fp)`. If any conversion or write fails, the function does not succeed.

basic_filebuf::pbackfail

Tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(int_type _Meta = traits_type::eof);
```

Parameters

_Meta

The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof`. Otherwise, it returns **`traits_type::not_eof`**(*_Meta*).

Remarks

The protected virtual member function puts back an element into the input buffer and then makes it the current element (pointed to by the next pointer). If *_Meta* == **`traits_type::eof`**, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **`ch = traits_type::to_char_type`**(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `ch`, it can decrement the next pointer for the input buffer.
- If the function can make a `putback` position available, it can do so, set the next pointer to point at that position, and store `ch` in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling `ungetc` for an element of type **`char`**.

basic_filebuf::pos_type

Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::pos_type pos_type;
```

`basic_filebuf::seekoff`

Tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(off_type _Off,  
    ios_base::seekdir _Way,  
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See [seekdir](#) for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf`< `Elem`, `Tr` >, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type `pos_type` stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either `pubseekoff` or `pubseekpos`. Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for [text streams](#), [binary streams](#), and [wide streams](#).

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fseek (fp, _Off, _Way)`. If that function succeeds and the resulting position `fposn` can be determined by calling `fgetpos (fp, &fposn)`, the function succeeds. If the function succeeds, it returns a value of type `pos_type` containing `fposn`. Otherwise, it returns an invalid stream position.

`basic_filebuf::seekpos`

Tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(pos_type _Sp, ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fsetpos (fp, &fposn)`, where `fposn` is the `fpos_t` object stored in `pos`. If that function succeeds, the function returns `pos`. Otherwise, it returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf< Elem, Tr>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type `pos_type` stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either `pubseekoff` or `pubseekpos`. Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams, binary streams, and wide streams.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls `overflow`. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet `fac` to call `fac.unshift` as needed. Each element `byte` of type `char` thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc (byte, fp)`. If the call to `fac.unshift` or any write fails, the function does not succeed.

basic_filebuf::setbuf

Performs an operation particular to each derived stream buffer.

```
virtual basic_streambuf<Elem, Tr> *setbuf(
    char_type* _Buffer,
    streamsize count);
```

Parameters

_Buffer

Pointer to a buffer.

count

Size of the buffer.

Return Value

The protected member function returns zero if the file pointer `fp` is a null pointer.

Remarks

`setbuf` calls `setvbuf (fp, (char *) _Buffer, _IOFBF, count * sizeof (Elem))` to offer the array of `count` elements beginning at *_Buffer* as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **this** to signal success.

basic_filebuf::swap

Exchanges the contents of this `basic_filebuf` for the contents of the provided `basic_filebuf`.

```
void swap(basic_filebuf& right);
```

Parameters

right

An `lvalue` reference to another `basic_filebuf`.

`basic_filebuf::sync`

Tries to synchronize the controlled streams with any associated external streams.

```
virtual int sync();
```

Return Value

Returns zero if the file pointer `fp` is a null pointer. Otherwise, it returns zero only if calls to both [overflow](#) and `fflush`(`fp`) succeed in flushing any pending output to the stream.

`basic_filebuf::traits_type`

Associates a type name with the `Tr` template parameter.

```
typedef Tr traits_type;
```

`basic_filebuf::underflow`

Extracts the current element from the input stream.

```
virtual int_type underflow();
```

Return Value

If the function cannot succeed, it returns **`traits_type::eof`**. Otherwise, it returns `ch`, converted as described in the Remarks section.

Remarks

The protected virtual member function endeavors to extract the current element `ch` from the input stream, and return the element as **`traits_type::to_int_type`**(`ch`). It can do so in various ways:

- If a read position is available, it takes `ch` as the element stored in the read position and advances the next pointer for the input buffer.
- It can read one or more elements of type **`char`**, as if by successive calls of the form `fgetc`(`fp`), and convert them to an element **`ch`** of type `Elem` by using the file conversion facet `fac` to call `fac.in` as needed. If any read or conversion fails, the function does not succeed.

See also

[<fstream>](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_fstream Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_filebuf` < `Elem` , `Tr` >, with elements of type `Elem` , whose character traits are determined by the class `Tr` .

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_fstream : public basic_iostream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually `char_traits` < `Elem` >).

Remarks

The object stores an object of class `basic_filebuf` < `Elem` , `Tr` >.

NOTE

The get pointer and put pointer of an fstream object are **NOT** independent of each other. If the get pointer moves, so does the put pointer.

Example

The following example demonstrates how to create a `basic_fstream` object that can be read from and written to.

```
// basic_fstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    fstream fs("fstream.txt", ios::in | ios::out | ios::trunc);
    if (!fs.bad())
    {
        // Write to the file.
        fs << "Writing to a basic_fstream object..." << endl;
        fs.close();

        // Dump the contents of the file to cout.
        fs.open("fstream.txt", ios::in);
        cout << fs.rdbuf();
        fs.close();
    }
}
```

```
Writing to a basic_fstream object...
```

Constructors

CONSTRUCTOR	DESCRIPTION
basic_fstream	Constructs an object of type <code>basic_fstream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer, of type pointer to basic_filebuf < <code>Elem</code> , <code>Tr</code> >.
swap	Exchanges the content of this object with the content of another <code>basic_fstream</code> object.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_fstream::basic_fstream`

Constructs an object of type `basic_fstream`.

```

basic_fstream();

explicit basic_fstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_fstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

basic_fstream(basic_fstream&& right);

```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the *shflag* parameter in [_fsopen](#), [_wfsopen](#).

Remarks

The first constructor initializes the base class by calling [basic_iostream](#)(`sb`), where `sb` is the stored object of class [basic_filebuf](#)< **Elem**, **Tr**>. It also initializes `sb` by calling [basic_filebuf](#)< **Elem**, **Tr**>.

The second and third constructors initialize the base class by calling [basic_iostream](#)(`sb`). It also initializes `sb` by calling [basic_filebuf](#)< **Elem**, **Tr**>, and then `sb.open(_Filename, _Mode)`. If the latter function returns a null pointer, the constructor calls [setstate](#)(`failbit`).

The fourth constructor initializes the object with the contents of `right`, treated as an rvalue reference.

Example

See [streampos](#) for an example that uses `basic_fstream`.

basic_fstream::close

Closes a file.

```

void close();

```

Remarks

The member function calls `rdbuf->close`.

Example

See [basic_filebuf::close](#) for an example of how to use `close`.

basic_fstream::is_open

Determines if a file is open.

```

bool is_open() const;

```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns `rdbuf->is_open`.

Example

See `basic_filebuf::is_open` for an example of how to use `is_open`.

basic_fstream::open

Opens a file.

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in `ios_base::openmode`.

_Prot

The default file opening protection, equivalent to the *shflag* parameter in `_fsopen`, `_wfsopen`.

Remarks

The member function calls `rdbuf -> open(_Filename, _Mode)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

Example

See `basic_filebuf::open` for an example of how to use `open`.

basic_fstream::operator=

Assigns to this object the content from a specified stream object. This is a move assignment that involves an rvalue that does not leave a copy behind.

```
basic_fstream& operator=(basic_fstream&& right);
```

Parameters

right

An lvalue reference to a `basic_fstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *right*, treated as an rvalue reference.

`basic_fstream::rdbuf`

Returns the address of the stored stream buffer, of type pointer to `basic_filebuf<Elem, Tr>`.

```
basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

The address of the stored stream buffer.

Example

See [basic_filebuf::close](#) for an example of how to use `rdbuf`.

`basic_fstream::swap`

Exchanges the contents of two `basic_fstream` objects.

```
void swap(basic_fstream& right);
```

Parameters

right

An lvalue reference to a `basic_fstream` object.

Remarks

The member function exchanges the contents of this object and the contents of *right*.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_ifstream Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_filebuf` < `Elem`, `Tr` >, with elements of type `Elem`, whose character traits are determined by the class `Tr`.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ifstream : public basic_istream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually `char_traits` < `Elem` >).

Remarks

The object stores an object of class `basic_filebuf` < `Elem`, `Tr` >.

Example

The following example shows how to read in text from a file.

```
// basic_ifstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    ifstream ifs("basic_ifstream_class.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
        ifs.close();
    }
}
```

Input: basic_ifstream_class.txt

This is the contents of basic_ifstream_class.txt.

Output

This is the contents of `basic_ifstream_class.txt`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>basic_ifstream</code>	Initializes a new instance of a <code>basic_ifstream</code> object.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>close</code>	Closes a file.
<code>is_open</code>	Determines if a file is open.
<code>open</code>	Opens a file.
<code>rdbuf</code>	Returns the address of the stored stream buffer.
<code>swap</code>	Exchanges the content of this <code>basic_ifstream</code> for the content of the provided <code>basic_ifstream</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns the content of this stream object. This is a move assignment involving an <code>rvalue</code> that does not leave a copy behind.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_ifstream::basic_ifstream`

Constructs an object of type `basic_ifstream`.

```
basic_ifstream();

explicit basic_ifstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_ifstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

basic_ifstream(basic_ifstream&& right);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the `shflag` parameter in [_fsopen](#), [_wfsopen](#).

Remarks

The first constructor initializes the base class by calling [basic_istream](#)(`sb`), where `sb` is the stored object of class [basic_filebuf](#)< `Elem`, `Tr` >. It also initializes `sb` by calling [basic_filebuf](#)< `Elem`, `Tr` >.

The second and third constructors initialize the base class by calling [basic_istream](#)(`sb`). It also initializes `sb` by calling [basic_filebuf](#)< `Elem`, `Tr` >, then `sb.open(_Filename, _Mode | ios_base::in)`. If the latter function returns a null pointer, the constructor calls **setstate**(`failbit`).

The fourth constructor initializes the object with the contents of `right`, treated as an rvalue reference.

Example

The following example shows how to read in text from a file. To create the file, see the example for [basic_ofstream::basic_ofstream](#).

```
// basic_ifstream_ctor.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    ifstream ifs("basic_ifstream_ctor.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
        ifs.close();
    }
}
```

basic_ifstream::close

Closes a file.

```
void close();
```

Remarks

The member function calls [rdbuf](#) -> [close](#).

Example

See [basic_filebuf::close](#) for an example that uses `close`.

basic_ifstream::is_open

Determines if a file is open.

```
bool is_open() const;
```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns [rdbuf](#) -> [is_open](#).

Example

See [basic_filebuf::is_open](#) for an example that uses `is_open`.

basic_ifstream::open

Opens a file.

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the `shflag` parameter in [_fsopen](#), [_wfsopen](#).

Remarks

The member function calls [rdbuf](#) -> [open](#)(*_Filename*, `_Mode` | **ios_base::in**). If open fails, the function calls [setstate](#)(`failbit`), which may throw an `ios_base::failure` exception.

Example

See [basic_filebuf::open](#) for an example that uses `open`.

basic_ifstream::operator=

Assigns the content of this stream object. This is a move assignment involving an rvalue that does not leave a copy behind.

```
basic_ifstream& operator=(basic_ifstream&& right);
```

Parameters

right

An rvalue reference to a `basic_ifstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *right*, treated as an rvalue reference. For more information, see [Lvalues and Rvalues](#).

basic_ifstream::rdbuf

Returns the address of the stored stream buffer.

```
basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

A pointer to a [basic_filebuf](#) object representing the stored stream buffer.

Example

See [basic_filebuf::close](#) for an example that uses `rdbuf`.

basic_ifstream::swap

Exchanges the contents of two `basic_ifstream` objects.

```
void swap(basic_ifstream& right);
```

Parameters

right

A reference to another stream buffer.

Remarks

The member function exchanges the contents of this object for the contents of *right*.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_ofstream Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_filebuf` < `Elem`, `Tr` >, with elements of type `Elem`, whose character traits are determined by the class `Tr`.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ofstream : public basic_ostream<Elem, Tr>
```

Parameters

Elem

The basic element of the file buffer.

Tr

The traits of the basic element of the file buffer (usually `char_traits` < `Elem` >).

Remarks

When the **wchar_t** specialization of `basic_ofstream` writes to the file, if the file is opened in text mode it will write a MBCS sequence. The internal representation will use a buffer of `wchar_t` characters.

The object stores an object of class `basic_filebuf` < `Elem`, `Tr` >.

Example

The following example shows how to create a `basic_ofstream` object and write text to it.

```
// basic_ofstream_class.cpp
// compile with: /EHsc
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

Constructors

CONSTRUCTOR	DESCRIPTION
<code>basic_ofstream</code>	Creates an object of type <code>basic_ofstream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer.
swap	Exchange the contents of this <code>basic_ofstream</code> for the contents of the provided <code>basic_ofstream</code> .

Operators

OPERATOR	DESCRIPTION
operator=	Assigns the content of this stream object. This is a move assignment involving an <code>rvalue reference</code> that does not leave a copy behind.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_ofstream::basic_ofstream`

Creates an object of type `basic_ofstream`.

```
basic_ofstream();

explicit basic_ofstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_ofstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

basic_ofstream(
    basic_ofstream&& right);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the `shf1ag` parameter in [_fsopen](#), [_wfsopen](#).

right

The rvalue reference to the `basic_ofstream` object being used to initialize this `basic_ofstream` object.

Remarks

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_filebuf< Elem, Tr >`. It also initializes `sb` by calling `basic_filebuf< Elem, Tr >`.

The second and third constructors initialize the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_filebuf< Elem, Tr >` and then `sb.open(_Filename, _Mode | ios_base::out)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

The fourth constructor is a copy function. It initializes the object with the contents of *right*, treated as an rvalue reference.

Example

The following example shows how to create a `basic_ofstream` object and write text to it.

```
// basic_ofstream_ctor.cpp
// compile with: /EHsc
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("C:\\\\ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

basic_ofstream::close

Closes a file.

```
void close();
```

Remarks

The member function calls `rdbuf->close`.

Example

See `basic_filebuf::close` for an example that uses `close`.

basic_ofstream::is_open

Indicates whether a file is open.

```
bool is_open() const;
```

Return Value

true if the file is open, **false** otherwise.

Remarks

The member function returns `rdbuf -> is_open`.

Example

```
// basic_ofstream_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
    using namespace std;
    ifstream file;
    // Open and close with a basic_filebuf
    file.rdbuf( )->open( "basic_ofstream_is_open.txt", ios::in );
    file.close( );
    if (file.is_open())
        cout << "it's open" << endl;
    else
        cout << "it's closed" << endl;
}
```

basic_ofstream::open

Opens a file.

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

_Filename

The name of the file to open.

_Mode

One of the enumerations in [ios_base::openmode](#).

_Prot

The default file opening protection, equivalent to the `shflag` parameter in [_fsopen](#), [_wfsopen](#).

Remarks

The member function calls `rdbuf -> open(_Filename, _Mode | ios_base::out)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

Example

See [basic_filebuf::open](#) for an example that uses `open`.

basic_ofstream::operator=

Assigns the content of this stream object. This is a move assignment involving an `rvalue reference` that does not leave a copy behind.

```
basic_ofstream& operator=(basic_ofstream&& right);
```

Parameters

right

An rvalue reference to a `basic_ofstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *right*, treated as an rvalue reference.

basic_ofstream::rdbuf

Returns the address of the stored stream buffer.

```
basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

Returns the address of the stored stream buffer.

Example

See [basic_filebuf::close](#) for an example that uses `rdbuf`.

basic_ofstream::swap

Exchanges the contents of two `basic_ofstream` objects.

```
void swap(basic_ofstream& right);
```

Parameters

right

An `lvalue` reference to another `basic_ofstream` object.

Remarks

The member function exchanges the contents of this object for the contents of *right*.

See also

[basic_ostream Class](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<functional>

2/28/2019 • 10 minutes to read • [Edit Online](#)

Defines C++ Standard Library functions that help construct *function objects*, also known as *functors*, and their binders. A function object is an object of a type that defines `operator()`. A function object can be a function pointer, but more typically, the object is used to store additional information that can be accessed during a function call.

Syntax

```
#include <functional>
```

Remarks

Algorithms require two types of function objects: *unary* and *binary*. Unary function objects require one argument, and binary function objects require two arguments. A function object and function pointers can be passed as a predicate to an algorithm, but function objects are also adaptable and increase the scope, flexibility, and efficiency of the C++ Standard Library. If, for example, a value needed to be bound to a function before being passed to an algorithm, then a function pointer could not be used. Function adaptors convert function pointers into adaptable function objects that can be bound to a value. The header `<functional>` also contains member function adaptors that allow member functions to be called as adaptable function objects. Functions are adaptable if they have nested type declarations specifying their argument and return types. Function objects and their adaptors allow the C++ Standard Library to upgrade existing applications and help integrate the library into the C++ programming environment.

The implementation of the function objects in `<functional>` includes *transparent operator functors*, which are specializations of standard function objects and take no template parameters, and perform perfect forwarding of the function arguments and perfect return of the result. These template specializations do not require that you specify argument types when you invoke arithmetic, comparison, logical, and bitwise operator functors. You can overload arithmetic, comparison, logical, or bitwise operators for your own types, or for heterogeneous combinations of types, and then use the transparent operator functors as function arguments. For example, if your type *MyType* implements `operator<`, you can call

```
sort(my_collection.begin(), my_collection.end(), less<>()) instead of explicitly specifying the type  
sort(my_collection.begin(), my_collection.end(), less<MyType>()).
```

The following features are added in C++11, C++14 and C++17:

- A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
- A *callable type* is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.
- A *callable object* is an object of a callable type.
- A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.
- A *call wrapper* is an object of a call wrapper type.
- A *target object* is the callable object held by a call wrapper object.

The pseudo-function `INVOKE(f, t1, t2, ..., tN)` means one of the following things:

- `(t1.*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`.
- `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is not one of the types described in the previous item.
- `t1.*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`.
- `(*t1).*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item.
- `f(t1, t2, ..., tN)` in all other cases.

The pseudo-function `INVOKE(f, t1, t2, ..., tN, R)` means `INVOKE(f, t1, t2, ..., tN)` implicitly converted to `R`.

If a call wrapper has a *weak result type*, the type of its member type `result_type` is based on the type `T` of the target object of the wrapper, as follows:

- If `T` is a pointer to function, `result_type` is a synonym for the return type of `T`.
- If `T` is a pointer to member function, `result_type` is a synonym for the return type of `T`.
- If `T` is a class type that has a member type `result_type`, then `result_type` is a synonym for `T::result_type`.
- Otherwise, there is no member `result_type`.

Every call wrapper has a move constructor and a copy constructor. A *simple call wrapper* is a call wrapper that has an assignment operator and whose copy constructor, move constructor, and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called by using an arbitrary argument list and that delivers the arguments to the wrapped callable object as references. All rvalue arguments are delivered as rvalue references, and lvalue arguments are delivered as lvalue references.

Classes

CLASS	DESCRIPTION
bad_function_call	A class that describes an exception thrown to indicate that a call to <code>operator()</code> on a function object failed because the object was empty.
binary_negate	A template class providing a member function that negates the return value of a specified binary function. (Deprecated in C++17.)
binder1st	A template class providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)

CLASS	DESCRIPTION
binder2nd	A template class providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
const_mem_fun_ref_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun1_ref_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun1_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
function	A class that wraps a callable object.
hash	A class that computes a hash code for a value.
is_bind_expression	A class that tests if a particular type is generated by calling <code>bind</code> .
is_placeholder	A class that tests if a particular type is a placeholder.
mem_fun_ref_t	An adapter class that allows a <code>non_const</code> member function that takes no arguments to be called as a unary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
mem_fun_t	An adapter class that allows a <code>non_const</code> member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
mem_fun1_ref_t	An adapter class that allows a <code>non_const</code> member function that takes a single argument to be called as a binary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
mem_fun1_t	An adapter class that allows a <code>non_const</code> member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)

CLASS	DESCRIPTION
pointer_to_binary_function	Converts a binary function pointer into an adaptable binary function. (Deprecated in C++11, removed in C++17.)
pointer_to_unary_function	Converts a unary function pointer into an adaptable unary function. (Deprecated in C++11, removed in C++17.)
reference_wrapper	A class that wraps a reference.
unary_negate	A template class providing a member function that negates the return value of a specified unary function. (Deprecated in C++17.)

Functions

FUNCTION	DESCRIPTION
bind	Binds arguments to a callable object.
bind1st	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the first argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
bind2nd	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the second argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
bit_and	Returns the bitwise logical AND (binary operator&) of the two parameters.
bit_not	Returns the bitwise logical complement (operator~) of the parameter. (Added in C++14.)
bit_or	Returns the bitwise logical OR (operator) of the two parameters.
bit_xor	Returns the bitwise logical XOR (operator^) of the two parameters.
cref	Constructs a const <code>reference_wrapper</code> from an argument.
mem_fn	Generates a simple call wrapper.
mem_fun	Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments. (Deprecated in C++11, removed in C++17.)

FUNCTION	DESCRIPTION
mem_fun_ref	A helper template function used to construct function object adaptors for member functions when initialized with reference arguments.
not1	Returns the complement of a unary predicate. (Deprecated in C++17.)
not2	Returns the complement of a binary predicate. (Deprecated in C++17.)
not_fn	Returns the complement of the result of its function object. (Added in C++17.)
ptr_fun	A helper template function used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions. (Deprecated in C++11, removed in C++17.)
ref	Constructs a <code>reference_wrapper</code> from an argument.
swap	Swaps two <code>function</code> objects.

Structs

STRUCT	DESCRIPTION
binary_function	An empty base class that defines types that may be inherited by derived class that provides a binary function object. (Deprecated in C++11, removed in C++17.)
divides	The class provides a predefined function object that performs the arithmetic operation of division on elements of a specified value type.
equal_to	A binary predicate that tests whether a value of a specified type is equal to another value of that type.
greater	A binary predicate that tests whether a value of a specified type is greater than another value of that type.
greater_equal	A binary predicate that tests whether a value of a specified type is greater than or equal to another value of that type.
less	A binary predicate that tests whether a value of a specified type is less than another value of that type.
less_equal	A binary predicate that tests whether a value of a specified type is less than or equal to another value of that type.
logical_and	The class provides a predefined function object that performs the logical operation of conjunction on elements of a specified value type and tests for the truth or falsity of the result.

STRUCT	DESCRIPTION
logical_not	The class provides a predefined function object that performs the logical operation of negation on elements of a specified value type and tests for the truth or falsity of the result.
logical_or	The class provides a predefined function object that performs the logical operation of disjunction on elements of a specified value type and tests for the truth or falsity of the result.
minus	The class provides a predefined function object that performs the arithmetic operation of subtraction on elements of a specified value type.
modulus	The class provides a predefined function object that performs the arithmetic operation of modulus on elements of a specified value type.
multiplies	The class provides a predefined function object that performs the arithmetic operation of multiplication on elements of a specified value type.
negate	The class provides a predefined function object that returns the negative of an element value.
not_equal_to	A binary predicate that tests whether a value of a specified type is not equal to another value of that type.
plus	The class provides a predefined function object that performs the arithmetic operation of addition on elements of a specified value type.
unary_function	An empty base class that defines types that may be inherited by derived class that provides a unary function object. (Deprecated in C++11, removed in C++17.)

Objects

OBJECT	DESCRIPTION
_1.._M	Placeholders for replaceable arguments.

Operators

OPERATOR	DESCRIPTION
operator==	Disallows equality comparison of callable objects.
operator!=	Disallows inequality comparison of callable objects.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<functional> functions

3/4/2019 • 27 minutes to read • [Edit Online](#)

bind	bit_and	bit_not
bit_or	bit_xor	cref
invoke	mem_fn	not_fn
ref	swap	

These functions are deprecated in C++11 and removed in C++17:

bind1st	bind2nd	mem_fun
mem_fun_ref	ptr_fun	

These functions are deprecated in C++17:

not1	not2

bind

Binds arguments to a callable object.

```
template <class FT, class T1, class T2, ..., class TN>
unspecified bind(FT fn, T1 t1, T2 t2, ..., TN tN);

template <class RTy, class FT, class T1, class T2, ..., class TN>
unspecified bind(FT fn, T1 t1, T2 t2, ..., TN tN);
```

Parameters

Fy

The type of the object to call.

TN

The type of the Nth call argument.

fn

The object to call.

tN

The Nth call argument.

Remarks

The types `FT, T1, T2, ..., TN` must be copy-constructible, and `INVOKE(fn, t1, ..., tN)` must be a valid

expression for some values `w1, w2, ..., wN`.

The first template function returns a forwarding call wrapper `g` with a weak result type. The effect of `g(u1, u2, ..., uM)` is `INVOKE(f, v1, v2, ..., vN, invoke_result<FT cv (V1, V2, ..., VN)>::type)`, where `cv` is the cv-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. You use it to bind arguments to a callable object to make a callable object with a tailored argument list.

The second template function returns a forwarding call wrapper `g` with a nested type `result_type` that is a synonym for `RTy`. The effect of `g(u1, u2, ..., uM)` is `INVOKE(f, v1, v2, ..., vN, RTy)`, where `cv` is the cv-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. You use it to bind arguments to a callable object to make a callable object with a tailored argument list and with a specified return type.

The values of the bound arguments `v1, v2, ..., vN` and their corresponding types `V1, V2, ..., VN` depend on the type of the corresponding argument `ti` of type `Ti` in the call to `bind` and the cv-qualifiers `cv` of the call wrapper `g` as follows:

if `ti` is of type `reference_wrapper<T>` the argument `vi` is `ti.get()` and its type `Vi` is `T&`;

if the value of `std::is_bind_expression<Ti>::value` is **true** the argument `vi` is `ti(u1, u2, ..., uM)` and its type `Vi` is `result_of<Ti cv (U1&, U2&, ..., UN&)>::type`;

if the value `j` of `std::is_placeholder<Ti>::value` isn't zero the argument `vi` is `uj` and its type `Vi` is `Uj&`;

otherwise the argument `vi` is `ti` and its type `Vi` is `Ti cv &`.

For example, given a function `f(int, int)` the expression `bind(f, _1, 0)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(x, 0)`. The expression `bind(f, 0, _1)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(0, x)`.

The number of arguments in a call to `bind` and the argument `fn` must be equal to the number of arguments that can be passed to the callable object `fn`. For example, `bind(cos, 1.0)` is correct, and both `bind(cos)` and `bind(cos, _1, 0.0)` are incorrect.

The number of arguments in the function call to the call wrapper returned by `bind` must be at least as large as the highest numbered value of `is_placeholder<PH>::value` for all of the placeholder arguments in the call to `bind`. For example, `bind(cos, _2)(0.0, 1.0)` is correct (and returns `cos(1.0)`), and `bind(cos, _2)(0.0)` is incorrect.

Example

```

// std_functional_bind.cpp
// compile with: /EHsc
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std::placeholders;

void square(double x)
{
    std::cout << x << "^2 == " << x * x << std::endl;
}

void product(double x, double y)
{
    std::cout << x << "*" << y << " == " << x * y << std::endl;
}

int main()
{
    double arg[] = { 1, 2, 3 };

    std::for_each(&arg[0], arg + 3, square);
    std::cout << std::endl;

    std::for_each(&arg[0], arg + 3, std::bind(product, _1, 2));
    std::cout << std::endl;

    std::for_each(&arg[0], arg + 3, std::bind(square, _1));

    return (0);
}

```

```

1^2 == 1
2^2 == 4
3^2 == 9

1*2 == 2
2*2 == 4
3*2 == 6

1^2 == 1
2^2 == 4
3^2 == 9

```

bind1st

A helper template function that creates an adaptor to convert a binary function object into a unary function object. It binds the first argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

```

template <class Operation, class Type>
binder1st <Operation> bind1st (const Operation& func, const Type& left);

```

Parameters

func

The binary function object to be converted to a unary function object.

left

The value to which the first argument of the binary function object is to be bound.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *left*.

Remarks

Function binders are a kind of function adaptor. Because they return function objects, they can be used in certain types of function composition to construct more complicated and powerful expressions.

If *func* is an object of type `Operation` and `c` is a constant, then `bind1st(func, c)` is the same as the [binder1st](#) class constructor `binder1st<Operation>(func, c)`, and is more convenient to use.

Example

```

// functional_bind1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan5: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 5);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind1st(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1a << "." << endl;

    // Compare: counting the number of integers > 5 in the vector
    // with a user defined function object
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan5());
    cout << "The number of elements in v1 greater than 5 is: "
        << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind2nd(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
        << result2 << "." << endl;
}

```

```

The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 5 is: 4.
The number of elements in v1 less than 10 is: 2.

```

bind2nd

A helper template function that creates an adaptor to convert a binary function object into a unary function object.

It binds the second argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

```
template <class Operation, class Type>
binder2nd <Operation> bind2nd(const Operation& func, const Type& right);
```

Parameters

func

The binary function object to be converted to a unary function object.

right

The value to which the second argument of the binary function object is to be bound.

Return Value

The unary function object result of binding the second argument of the binary function object to *right*.

Remarks

Function binders are a kind of function adaptor. Because they return function objects, they can be used in certain types of function composition to construct more complicated and powerful expressions.

If *func* is an object of type `Operation` and `c` is a constant, then `bind2nd(func, c)` is the same as the [binder2nd](#) class constructor `binder2nd<Operation>(func, c)`, and more convenient to use.

Example

```

// functional_bind2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan15: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 15);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1a << "." << endl;

    // Compare counting the number of integers > 15 in the vector
    // with a user-defined function object
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan15());
    cout << "The number of elements in v1 greater than 15 is: "
        << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind1st(greater<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
        << result2 << "." << endl;
}

```

```

The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 15 is: 2.
The number of elements in v1 less than 10 is: 2.

```


A predefined function object that does a bitwise AND operation (binary `operator&`) on its arguments.

```
template <class Type = void>
struct bit_and : public binary_function<Type, Type, Type> {
    Type operator()(
        const Type& Left,
        const Type& Right) const;
};

// specialized transparent functor for operator&
template <>
struct bit_and<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const ->
        decltype(std::forward<T>(Left) & std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator&` that takes operands of the specified or inferred types.

Left

The left operand of the bitwise AND operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the bitwise AND operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left & Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator&`.

Remarks

The `bit_and` functor is restricted to integral types for the basic data types, or to user-defined types that implement binary `operator&`.

bit_not

A predefined function object that does a bitwise complement (NOT) operation (unary `operator~`) on its argument. Added in C++14.

```
template <class Type = void>
struct bit_not : public unary_function<Type, Type>
{
    Type operator()(const Type& Right) const;
};

// specialized transparent functor for operator~
template <>
struct bit_not<void>
{
    template <class Type>
    auto operator()(Type&& Right) const -> decltype(~std::forward<Type>(Right));
};
```

Parameters

Type

A type that supports a unary `operator~`.

Right

The operand of the bitwise complement operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of an lvalue or rvalue reference argument of inferred type *Type*.

Return Value

The result of `~ Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator~`.

Remarks

The `bit_not` functor is restricted to integral types for the basic data types, or to user-defined types that implement binary `operator~`.

bit_or

A predefined function object that does a bitwise OR operation (`operator|`) on its arguments.

```
template <class Type = void>
struct bit_or : public binary_function<Type, Type, Type> {
    Type operator()(
        const Type& Left,
        const Type& Right) const;
};

// specialized transparent functor for operator|
template <>
struct bit_or<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) | std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator|` that takes operands of the specified or inferred types.

Left

The left operand of the bitwise OR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the bitwise OR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left | Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator|`.

Remarks

The `bit_or` functor is restricted to integral types for the basic data types, or to user-defined types that implement

operator| .

bit_xor

A predefined function object that does a bitwise XOR operation (binary `operator^`) on its arguments.

```
template <class Type = void>
struct bit_xor : public binary_function<Type, Type, Type> {
    Type operator()(
        const Type& Left,
        const Type& Right) const;
};

// specialized transparent functor for operator^
template <>
struct bit_xor<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) ^ std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator^` that takes operands of the specified or inferred types.

Left

The left operand of the bitwise XOR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the bitwise XOR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left ^ Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator^`.

Remarks

The `bit_xor` functor is restricted to integral types for the basic data types, or to user-defined types that implement binary `operator^`.

cref

Constructs a const `reference_wrapper` from an argument.

```
template <class Ty>
reference_wrapper<const Ty> cref(const Ty& arg);

template <class Ty>
reference_wrapper<const Ty> cref(const reference_wrapper<Ty>& arg);
```

Parameters

Ty

The type of the argument to wrap.

arg

The argument to wrap.

Remarks

The first function returns `reference_wrapper<const Ty>(arg.get())`. You use it to wrap a const reference. The second function returns `reference_wrapper<const Ty>(arg)`. You use it to rewrap a wrapped reference as a const reference.

Example

```
// std__functional__cref.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    int i = 1;

    std::cout << "i = " << i << std::endl;
    std::cout << "cref(i) = " << std::cref(i) << std::endl;
    std::cout << "cref(neg)(i) = "
        << std::cref(neg)(i) << std::endl;

    return (0);
}
```

```
i = 1
cref(i) = 1
cref(neg)(i) = -1
```

invoke

Invokes any callable object with the given arguments. Added in C++17.

```
template <class Callable, class... Args>
invoke_result_t<Callable, Args...>
    invoke(Callable&& fn, Args&&... args) noexcept(/* specification */);
```

Parameters

Callable

The type of the object to call.

Args

The types of the call arguments.

fn

The object to call.

args

The call arguments.

specification

The **noexcept** specification `std::is_nothrow_invocable_v<Callable, Args>`.

Remarks

Invokes the callable object *fn* using the parameters *args*. Effectively,

`INVOKE(std::forward<Callable>(fn), std::forward<Args>(args)...) ,` where the pseudo-function

`INVOKE(f, t1, t2, ..., tN)` means one of the following things:

- `(t1.*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`. That is, when `std::is_base_of<T, std::decay_t<decltype(t1)>>::value` is true.
- `(t1.get().*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `std::decay_t<decltype(t1)>` is a specialization of `std::reference_wrapper`.
- `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` isn't one of the previous types.
- `t1.*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`. That is, when `std::is_base_of<T, std::decay_t<decltype(t1)>>::value` is true.
- `t1.get().*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `std::decay_t<decltype(t1)>` is a specialization of `std::reference_wrapper`.
- `(*t1).*f` when `N == 1` and `f` is a pointer to member data of a class `T` and `t1` isn't one of the previous types.
- `f(t1, t2, ..., tN)` in all other cases.

For information on the result type of a callable object, see [invoke_result](#). For predicates on callable types, see [is_invocable](#), [is_invocable_r](#), [is_nothrow_invocable](#), [is_nothrow_invocable_r](#) classes.

Example

```

// functional_invoke.cpp
// compile using: cl /EHsc /std:c++17 functional_invoke.cpp
#include <functional>
#include <iostream>

struct Demo
{
    int n_;

    Demo(int const n) : n_{n} {}

    void operator()( int const i, int const j ) const
    {
        std::cout << "Demo operator( " << i << ", "
                    << j << " ) is " << i * j << "\n";
    }

    void difference( int const i ) const
    {
        std::cout << "Demo.difference( " << i << " ) is "
                    << n_ - i << "\n";
    }
};

void divisible_by_3(int const i)
{
    std::cout << i << ( i % 3 == 0 ? " is" : " isn't" )
                << " divisible by 3.\n";
}

int main()
{
    Demo d{ 42 };
    Demo * pd{ &d };
    auto pmf = &Demo::difference;
    auto pmd = &Demo::n_;

    // Invoke a function object, like calling d( 3, -7 )
    std::invoke( d, 3, -7 );

    // Invoke a member function, like calling
    // d.difference( 29 ) or (d.*pmf)( 29 )
    std::invoke( &Demo::difference, d, 29 );
    std::invoke( pmf, pd, 13 );

    // Invoke a data member, like access to d.n_ or d.*pmd
    std::cout << "d.n_: " << std::invoke( &Demo::n_, d ) << "\n";
    std::cout << "pd->n_: " << std::invoke( pmd, pd ) << "\n";

    // Invoke a stand-alone (free) function
    std::invoke( divisible_by_3, 42 );

    // Invoke a lambda
    auto divisible_by_7 = [] ( int const i ) {
        std::cout << i << ( i % 7 == 0 ? " is" : " isn't" )
                    << " divisible by 7.\n";
    };
    std::invoke( divisible_by_7, 42 );
}

```

```
Demo operator( 3, -7 ) is -21
Demo.difference( 29 ) is 13
Demo.difference( 13 ) is 29
d.n_: 42
pd->n_: 42
42 is divisible by 3.
42 is divisible by 7.
```

mem_fn

Generates a simple call wrapper.

```
template <class RTy, class Ty>
unspecified mem_fn(RTy Ty::*pm);
```

Parameters

RTy

The return type of the wrapped function.

Ty

The type of the member function pointer.

Remarks

The template function returns a simple call wrapper `cw`, with a weak result type, such that the expression `cw(t, a2, ..., aN)` is the same as `INVOKE(pm, t, a2, ..., aN)`. It doesn't throw any exceptions.

The returned call wrapper is derived from `std::unary_function<cv Ty*, RTy>` (and defining the nested type `result_type` as a synonym for `RTy` and the nested type `argument_type` as a synonym for `cv Ty*`) only if the type `Ty` is a pointer to member function with cv-qualifier `cv` that takes no arguments.

The returned call wrapper is derived from `std::binary_function<cv Ty*, T2, RTy>` (and defining the nested type `result_type` as a synonym for `RTy`, the nested type `first argument_type` as a synonym for `cv Ty*`, and the nested type `second argument_type` as a synonym for `T2`) only if the type `Ty` is a pointer to member function with cv-qualifier `cv` that takes one argument, of type `T2`.

Example

```

// std_functional_mem_fn.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

class Funs
{
public:
    void square(double x)
    {
        std::cout << x << "^2 == " << x * x << std::endl;
    }

    void product(double x, double y)
    {
        std::cout << x << "*" << y << " == " << x * y << std::endl;
    }
};

int main()
{
    Funs funs;

    std::mem_fn(&Funs::square)(funs, 3.0);
    std::mem_fn(&Funs::product)(funs, 3.0, 2.0);

    return (0);
}

```

```

3^2 == 9
3*2 == 6

```

mem_fun

Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments. Deprecated in C++11 for [mem_fn](#) and [bind](#), and removed in C++17.

```

template <class Result, class Type>
mem_fun_t<Result, Type> mem_fun (Result(Type::* pMem)());

template <class Result, class Type, class Arg>
mem_fun1_t<Result, Type, Arg> mem_fun(Result (Type::* pMem)(Arg));

template <class Result, class Type>
const_mem_fun_t<Result, Type> mem_fun(Result (Type::* pMem)() const);

template <class Result, class Type, class Arg>
const_mem_fun1_t<Result, Type, Arg> mem_fun(Result (Type::* pMem)(Arg) const);

```

Parameters

pMem

A pointer to the member function of class `Type` to be converted to a function object.

Return Value

A **const** or **non_const** function object of type `mem_fun_t` or `mem_fun1_t`.

Example


```

// functional_mem_fun.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class StoreVals
{
    int val;
public:
    StoreVals() { val = 0; }
    StoreVals(int j) { val = j; }

    bool display() { cout << val << " "; return true; }
    int squareval() { val *= val; return val; }
    int lessconst(int k) {val -= k; return val; }
};

int main( )
{
    vector<StoreVals *> v1;

    StoreVals sv1(5);
    v1.push_back(&sv1);
    StoreVals sv2(10);
    v1.push_back(&sv2);
    StoreVals sv3(15);
    v1.push_back(&sv3);
    StoreVals sv4(20);
    v1.push_back(&sv4);
    StoreVals sv5(25);
    v1.push_back(&sv5);

    cout << "The original values stored are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;

    // Use of mem_fun calling member function through a pointer
    // square each value in the vector using squareval ()
    for_each(v1.begin(), v1.end(), mem_fun<int, StoreVals>(&StoreVals::squareval));
    cout << "The squared values are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;

    // Use of mem_fun1 calling member function through a pointer
    // subtract 5 from each value in the vector using lessconst ()
    for_each(v1.begin(), v1.end(),
        bind2nd (mem_fun1<int, StoreVals,int>(&StoreVals::lessconst), 5));
    cout << "The squared values less 5 are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>(&StoreVals::display));
    cout << endl;
}

```

mem_fun_ref

Helper template functions used to construct function object adaptors for member functions when initialized by using reference arguments. Deprecated in C++11, removed in C++17.

```

template <class Result, class Type>
mem_fun_ref_t<Result, Type> mem_fun_ref(Result (Type::* pMem)());

template <class Result, class Type, class Arg>
mem_fun1_ref_t<Result, Type, Arg> mem_fun_ref(Result (Type::* pMem)(Arg));

template <class Result, class Type>
const_mem_fun_ref_t<Result, Type> mem_fun_ref(Result Type::* pMem]() const);

template <class Result, class Type, class Arg>
const_mem_fun1_ref_t<Result, Type, Arg> mem_fun_ref(Result (T::* pMem)(Arg) const);

```

Parameters

pMem

A pointer to the member function of class `Type` to be converted to a function object.

Return Value

A **const** or `non_const` function object of type `mem_fun_ref_t` or `mem_fun1_ref_t`.

Example

```

// functional_mem_fun_ref.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class NumVals
{
    int val;
public:
    NumVals ( ) { val = 0; }
    NumVals ( int j ) { val = j; }

    bool display ( ) { cout << val << " "; return true; }
    bool isEven ( ) { return ( bool ) !( val %2 ); }
    bool isPrime( )
    {
        if (val < 2) { return true; }
        for (int i = 2; i <= val / i; ++i)
        {
            if (val % i == 0) { return false; }
        }
        return true;
    }
};

int main( )
{
    vector <NumVals> v1 ( 13 ), v2 ( 13 );
    vector <NumVals>::iterator v1_Iter, v2_Iter;
    int i, k;

    for ( i = 0; i < 13; i++ ) v1 [ i ] = NumVals ( i+1 );
    for ( k = 0; k < 13; k++ ) v2 [ k ] = NumVals ( k+1 );

    cout << "The original values stored in v1 are: " ;
    for_each( v1.begin( ), v1.end( ),
        mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    // Use of mem_fun_ref calling member function through a reference
    // remove the primes in the vector using isPrime ( )
    v1_Iter = remove_if ( v1.begin( ), v1.end( ),
        mem_fun_ref ( &NumVals::isPrime ) );
    cout << "With the primes removed, the remaining values in v1 are: " ;
    for_each( v1.begin( ), v1_Iter,
        mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    cout << "The original values stored in v2 are: " ;
    for_each( v2.begin( ), v2.end( ),
        mem_fun_ref ( &NumVals::display ) );
    cout << endl;

    // Use of mem_fun_ref calling member function through a reference
    // remove the even numbers in the vector v2 using isEven ( )
    v2_Iter = remove_if ( v2.begin( ), v2.end( ),
        mem_fun_ref ( &NumVals::isEven ) );
    cout << "With the even numbers removed, the remaining values are: " ;
    for_each( v2.begin( ), v2_Iter,
        mem_fun_ref ( &NumVals::display ) );
    cout << endl;
}

```

```
The original values stored in v1 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the primes removed, the remaining values in v1 are: 4 6 8 9 10 12
The original values stored in v2 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the even numbers removed, the remaining values are: 1 3 5 7 9 11 13
```

not1

Returns the complement of a unary predicate. Deprecated for [not_fn](#) in C++17.

```
template <class UnaryPredicate>
unary_negate<UnaryPredicate> not1(const UnaryPredicate& predicate);
```

Parameters

predicate

The unary predicate to be negated.

Return Value

A unary predicate that is the negation of the unary predicate modified.

Remarks

If a `unary_negate` is constructed from a unary predicate `predicate(x)`, then it returns `!predicate(x)`.

Example

```

// functional_not1.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        not1(bind2nd(greater<int>(), 10)));

    cout << "The number of elements in v1 not greater than 10 is: "
        << result2 << "." << endl;
}

```

```

The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.

```

not2

Returns the complement of a binary predicate. Deprecated for [not_fn](#) in C++17.

```

template <class BinaryPredicate>
binary_negate<BinaryPredicate> not2(const BinaryPredicate& func);

```

Parameters

func

The binary predicate to be negated.

Return Value

A binary predicate that is the negation of the binary predicate modified.

Remarks

If a `binary_negate` is constructed from a binary predicate `binary_predicate(x, y)`, then it returns `!binary_predicate(x, y)`.

Example

```
// functional_not2.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    v1.push_back( 6262 );
    v1.push_back( 6262 );
    for ( i = 0 ; i < 5 ; i++ )
    {
        v1.push_back( rand( ) );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    // use default binary predicate less<int>( )
    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order,
    // use the binary_negate helper function not2
    sort( v1.begin( ), v1.end( ), not2(less<int>( ) ) );
    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

```
Original vector v1 = ( 6262 6262 41 18467 6334 26500 19169 )
Sorted vector v1 = ( 41 6262 6262 6334 18467 19169 26500 )
Resorted vector v1 = ( 26500 19169 18467 6334 6262 6262 41 )
```

not_fn

The `not_fn` function template takes a callable object and returns a callable object. When the returned callable object is later invoked with some arguments, it passes them to the original callable object, and logically negates the result. It preserves the const qualification and value category behavior of the wrapped callable object. `not_fn` is new in C++17, and replaces the deprecated `std::not1`, `std::not2`, `std::unary_negate`, and `std::binary_negate`.

```
template <class Callable>
/* unspecified */ not_fn(Callable&& func);
```

Parameters

func

A callable object used to construct the forwarding call wrapper.

Remarks

The template function returns a call wrapper like `return call_wrapper(std::forward<Callable>(func))`, based on this exposition-only class:

```
class call_wrapper
{
    using FD = decay_t<Callable>;
    explicit call_wrapper(Callable&& func);

public:
    call_wrapper(call_wrapper&&) = default;
    call_wrapper(call_wrapper const&) = default;

    template<class... Args>
        auto operator()(Args&&...) & -> decltype(!declval<invoke_result_t<FD&(Args...)>>());

    template<class... Args>
        auto operator()(Args&&...) const& -> decltype(!declval<invoke_result_t<FD const&(Args...)>>());

    template<class... Args>
        auto operator()(Args&&...) && -> decltype(!declval<invoke_result_t<FD(Args...)>>());

    template<class... Args>
        auto operator()(Args&&...) const&& -> decltype(!declval<invoke_result_t<FD const(Args...)>>());

private:
    FD fd;
};
```

The explicit constructor on the callable object *func* requires type `std::decay_t<Callable>` to satisfy the requirements of `MoveConstructible`, and `is_constructible_v<FD, Callable>` must be true. It initializes the wrapped callable object `fd` from `std::forward<Callable>(func)`, and throws any exception thrown by construction of `fd`.

The wrapper exposes call operators distinguished by lvalue or rvalue reference category and const qualification as shown here:

```
template<class... Args> auto operator()(Args&&... args) & -> decltype(!declval<invoke_result_t<FD&(Args...)>>());
template<class... Args> auto operator()(Args&&... args) const& -> decltype(!declval<invoke_result_t<FD const&(Args...)>>());
template<class... Args> auto operator()(Args&&... args) && -> decltype(!declval<invoke_result_t<FD(Args...)>>());
template<class... Args> auto operator()(Args&&... args) const&& -> decltype(!declval<invoke_result_t<FD const(Args...)>>());
```

The first two are the same as `return !std::invoke(fd, std::forward<Args>(args)...) .` The second two are the same as `return !std::invoke(std::move(fd), std::forward<Args>(args)...) .`

Example

```

// functional_not_fn_.cpp
// compile with: /EHsc /std:c++17
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main()
{
    std::vector<int> v1 = { 99, 6264, 41, 18467, 6334, 26500, 19169 };
    auto divisible_by_3 = [](int i){ return i % 3 == 0; };

    std::cout << "Vector v1 = ( " ;
    for (const auto& item : v1)
    {
        std::cout << item << " ";
    }
    std::cout << ")" << std::endl;

    // Count the number of vector elements divisible by 3.
    int divisible =
        std::count_if(v1.begin(), v1.end(), divisible_by_3);
    std::cout << "Elements divisible by three: "
        << divisible << std::endl;

    // Count the number of vector elements not divisible by 3.
    int not_divisible =
        std::count_if(v1.begin(), v1.end(), std::not_fn(divisible_by_3));
    std::cout << "Elements not divisible by three: "
        << not_divisible << std::endl;
}

```

```

Vector v1 = ( 99 6264 41 18467 6334 26500 19169 )
Elements divisible by three: 2
Elements not divisible by three: 5

```

ptr_fun

Helper template functions used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions. Deprecated in C++11, removed in C++17.

```

template <class Arg, class Result>
pointer_to_unary_function<Arg, Result, Result (*)(Arg)> ptr_fun(Result (*pfunc)(Arg));

template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result, Result (*)(Arg1, Arg2)> ptr_fun(Result (*pfunc)(Arg1, Arg2));

```

Parameters

pfunc

The unary or binary function pointer to be converted to an adaptable function.

Return Value

The first template function returns the unary function [pointer_to_unary_function](#) < `Arg`, `Result`>(* `pfunc`).

The second template function returns binary function [pointer_to_binary_function](#) < `Arg1`, `Arg2`, `Result`>(* `pfunc`).

Remarks

A function pointer is a function object. It may be passed to any algorithm that expects a function as a parameter, but it isn't adaptable. Information about its nested types is required to use it with an adaptor, for example, to bind a

value to it or to negate it. The conversion of unary and binary function pointers by the `ptr_fun` helper function allows the function adaptors to work with unary and binary function pointers.

Example

```
// functional_ptr_fun.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
#include <iostream>

int main( )
{
    using namespace std;
    vector <char*> v1;
    vector <char*>::iterator Iter1, RIter;

    v1.push_back ( "Open" );
    v1.push_back ( "up" );
    v1.push_back ( "the" );
    v1.push_back ( "opalescent" );
    v1.push_back ( "gates" );

    cout << "Original sequence contains: " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; ++Iter1 )
        cout << *Iter1 << " ";
    cout << endl;

    // To search the sequence for "opalescent"
    // use a pointer_to_function conversion
    RIter = find_if( v1.begin( ), v1.end( ),
        not1 ( bind2nd ( ptr_fun ( strcmp ), "opalescent" ) ) );

    if ( RIter != v1.end( ) )
    {
        cout << "Found a match: "
            << *RIter << endl;
    }
}
```

ref

Constructs a `reference_wrapper` from an argument.

```
template <class Ty>
reference_wrapper<Ty> ref(Ty& arg);

template <class Ty>
reference_wrapper<Ty> ref(reference_wrapper<Ty>& arg);
```

Return Value

A reference to `arg`; specifically, `reference_wrapper<Ty>(arg)`.

Example

The following example defines two functions: one bound to a string variable, the other bound to a reference of the string variable computed by a call to `ref`. When the value of the variable changes, the first function continues to use the old value and the second function uses the new value.

```

#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <ostream>
#include <string>
#include <vector>
using namespace std;
using namespace std;
using namespace std::placeholders;

bool shorter_than(const string& l, const string& r) {
    return l.size() < r.size();
}

int main() {
    vector<string> v_original;
    v_original.push_back("tiger");
    v_original.push_back("cat");
    v_original.push_back("lion");
    v_original.push_back("cougar");

    copy(v_original.begin(), v_original.end(), ostream_iterator<string>(cout, " "));
    cout << endl;

    string s("meow");

    function<bool (const string&)> f = bind(shorter_than, _1, s);
    function<bool (const string&)> f_ref = bind(shorter_than, _1, ref(s));

    vector<string> v;

    // Remove elements that are shorter than s ("meow")

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f), v.end());

    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;

    // Now change the value of s.
    // f_ref, which is bound to ref(s), will use the
    // new value, while f is still bound to the old value.

    s = "kitty";

    // Remove elements that are shorter than "meow" (f is bound to old value of s)

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f), v.end());

    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;

    // Remove elements that are shorter than "kitty" (f_ref is bound to ref(s))

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f_ref), v.end());

    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;
}

```

```
tiger cat lion cougar
tiger lion cougar
tiger lion cougar
tiger cougar
```

swap

Swaps two `function` objects.

```
template <class FT>
void swap(function<FT>& f1, function<FT>& f2);
```

Parameters

FT

The type controlled by the function objects.

f1

The first function object.

f2

The second function object.

Remarks

The function returns `f1.swap(f2)` .

Example

```
// std_functional_swap.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << std::endl;

    swap(fn0, fn1);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
}
```

```
empty == false  
val == -3  
empty == true  
  
empty == true  
empty == false  
val == -3
```

See also

[<functional>](#)

<functional> operators

11/9/2018 • 2 minutes to read • [Edit Online](#)

`operator!=`

`operator==`

operator==

Tests if callable object is empty.

```
template <class Fty>
bool operator==(const function<Fty>& f, nullptr_type npc);

template <class Fty>
bool operator==(nullptr_type npc, const function<Fty>& f);
```

Parameters

Fty

The function type to wrap.

f

The function object

npc

A null pointer.

Remarks

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer constant. Both return true only if the `function` object is empty.

Example

```
// std_functional_operator_eq.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int(int)> fn0;
    std::cout << std::boolalpha << "empty == "
              << (fn0 == 0) << std::endl;

    std::function<int(int)> fn1(neg);
    std::cout << std::boolalpha << "empty == "
              << (fn1 == 0) << std::endl;

    return (0);
}
```

```
empty == true
empty == false
```

operator!=

Tests if callable object is not empty.

```
template <class Fty>
bool operator!=(const function<Fty>& f, null_ptr_type npc);

template <class Fty>
bool operator!=(null_ptr_type npc, const function<Fty>& f);
```

Parameters

Fty

The function type to wrap.

f

The function object

npc

A null pointer.

Remarks

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer constant. Both return true only if the `function` object is not empty.

Example

```
// std_functional_operator_ne.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0;
    std::cout << std::boolalpha << "not empty == "
              << (fn0 != 0) << std::endl;

    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "not empty == "
              << (fn1 != 0) << std::endl;

    return (0);
}
```

```
not empty == false
not empty == true
```

See also

<functional>

_1 Object

11/9/2018 • 2 minutes to read • [Edit Online](#)

Placeholders for replaceable arguments.

Syntax

```
namespace placeholders {  
    extern unspecified _1,  
    _2, ... _M  
} // namespace placeholders (within std)
```

Remarks

The objects `_1, _2, ... _M` are placeholders designating the first, second, ..., Mth argument, respectively in a function call to an object returned by `bind`. You use `_N` to specify where the Nth argument should be inserted when the bind expression is evaluated.

In this implementation the value of `_M` is 20.

Example

```
// std__functional_placeholder.cpp  
// compile with: /EHsc  
#include <functional>  
#include <algorithm>  
#include <iostream>  
  
using namespace std::placeholders;  
  
void square(double x)  
{  
    std::cout << x << "^2 == " << x * x << std::endl;  
}  
  
void product(double x, double y)  
{  
    std::cout << x << "*" << y << " == " << x * y << std::endl;  
}  
  
int main()  
{  
    double arg[] = {1, 2, 3};  
  
    std::for_each(&arg[0], &arg[3], square);  
    std::cout << std::endl;  
  
    std::for_each(&arg[0], &arg[3], std::bind(product, _1, 2));  
    std::cout << std::endl;  
  
    std::for_each(&arg[0], &arg[3], std::bind(square, _1));  
  
    return (0);  
}
```



```
1^2 == 1
2^2 == 4
3^2 == 9

1*2 == 2
2*2 == 4
3*2 == 6

1^2 == 1
2^2 == 4
3^2 == 9
```

Requirements

Header: <functional>

Namespace: std

See also

[bind](#)

[is_placeholder Class](#)

bad_function_call Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Reports a bad function call.

Syntax

```
class bad_function_call : public std::exception {};
```

Remarks

The class describes an exception thrown to indicate that a call to `operator()` on a [function Class](#)

binary_function Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

An empty base struct that defines types that may be inherited by derived classes that provides a binary function object. Deprecated in C++11, removed in C++17.

Syntax

```
struct binary_function {  
    typedef Arg1 first_argument_type;  
    typedef Arg2 second_argument_type;  
    typedef Result result_type;  
};
```

Remarks

The template struct serves as a base for classes that define a member function of the form:

```
result_type ** operator()( const** first_argument_type&, const second_argument_type& ) const
```

All such binary functions can refer to their first argument type as *first_argument_type*, their second argument type as *second_argument_type*, and their return type as *result_type*.

Example

```

// functional_binary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

template <class Type> class average:
binary_function<Type, Type, Type>
{
public:
    result_type operator( ) ( first_argument_type a,
                              second_argument_type b )
    {
        return (result_type) ( ( a + b ) / 2 );
    }
};

int main( )
{
    vector <double> v1, v2, v3 ( 6 );
    vector <double>::iterator Iter1, Iter2, Iter3;

    for ( int i = 1 ; i <= 6 ; i++ )
        v1.push_back( 11.0 / i );

    for ( int j = 0 ; j <= 5 ; j++ )
        v2.push_back( -2.0 * j );

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise averages of the elements of v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
                average<double>( ) );

    cout << "The element-wise averages are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 11 5.5 3.66667 2.75 2.2 1.83333 )
The vector v2 = ( -0 -2 -4 -6 -8 -10 )
The element-wise averages are: ( 5.5 1.75 -0.166667 -1.625 -2.9 -4.08333 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

binary_negate Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

A template class providing a member function that negates the return value of a specified binary function. Deprecated in C++17 in favor of [not_fn](#).

Syntax

```
template <class Operation>
class binary_negate
    : public binaryFunction <typename Operation::first_argument_type,
                           typename Operation::second_argument_type, bool>
{
public:
    explicit binary_negate(const Operation& Func);
    bool operator()(const typename Operation::first_argument_type& left,
                   const typename Operation::second_argument_type& right) const;
};
```

Parameters

Func

The binary function to be negated.

left

The left operand of the binary function to be negated.

right

The right operand of the binary function to be negated.

Return Value

The negation of the binary function.

Remarks

The template class stores a copy of a binary function object *Func*. It defines its member function `operator()` as returning `!Func(left, right)`.

The constructor of `binary_negate` is rarely used directly. The helper function [not2](#) is usually preferred to declare and use the **binary_negator** adaptor predicate.

Example

```

// functional_binary_negate.cpp
// compile with: /EHsc
#define _CRT_RAND_S
#include <stdlib.h>

#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;
    vector<unsigned int> v1;
    vector<unsigned int>::iterator Iter1;

    unsigned int i;
    v1.push_back( 6262 );
    v1.push_back( 6262 );
    unsigned int randVal = 0;
    for ( i = 0 ; i < 5 ; i++ )
    {
        rand_s(&randVal);
        v1.push_back( randVal );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    // use default binary predicate less<unsigned int>( )
    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order,
    // use the binary_negate function
    sort( v1.begin( ), v1.end( ),
        binary_negate<less<unsigned int>>(less<unsigned int>( ) ) );

    // The helper function not2 could also have been used
    // in the above line and is usually preferred for convenience
    // sort( v1.begin( ), v1.end( ), not2(less<unsigned int>( ) ) );

    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
/* Output:
Original vector v1 = ( 6262 6262 2233879413 2621500314 580942933 3715465425 3739828298 )
Sorted vector v1 = ( 6262 6262 580942933 2233879413 2621500314 3715465425 3739828298 )
Resorted vector v1 = ( 3739828298 3715465425 2621500314 2233879413 580942933 6262 6262 )
*/

```

Requirements

Header: <functional>

std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

binder1st Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

A template class providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value. Deprecated in C++11 in favor of [bind](#), and removed in C++17.

Syntax

```
template <class Operation>
class binder1st
    : public unaryFunction <typename Operation::second_argument_type,
                          typename Operation::result_type>
{
public:
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;
    binder1st(
        const Operation& binary_fn,
        const typename Operation::first_argument_type& left);

    result_type operator()(const argument_type& right) const;
    result_type operator()(const argument_type& right) const;

protected:
    Operation op;
    typename Operation::first_argument_type value;
};
```

Parameters

binary_fn

The binary function object to be converted to a unary function object.

left

The value to which the first argument of the binary function object is to be bound.

right

The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *left*.

Remarks

The template class stores a copy of a binary function object *binary_fn* in `op`, and a copy of *left* in `value`. It defines its member function `operator()` as returning `op(value, right)`.

If *binary_fn* is an object of type `Operation` and `c` is a constant, then `bind1st(binary_fn, c)` is a more convenient equivalent to `binder1st<Operation>(binary_fn, c)`. For more information, see [bind1st](#).

Example

```

// functional_binder1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
        binder1st<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;

    // Compare use of binder2nd fixing 2nd argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder2nd<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
        << result2 << "." << endl;
}
/* Output:
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

binder2nd Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

A template class providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Operation>
class binder2nd
    : public unaryFunction <typename Operation::first_argument_type,
        typename Operation::result_type>
{
public:
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;
    binder2nd(
        const Operation& Func,
        const typename Operation::second_argument_type& right);

    result_type operator()(const argument_type& left) const;
    result_type operator()(argument_type& left) const;

protected:
    Operation op;
    typename Operation::second_argument_type value;
};
```

Parameters

Func

The binary function object to be converted to a unary function object.

right

The value to which the second argument of the binary function object is to be bound.

left

The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return Value

The unary function object that results from binding the second argument of the binary function object to the value *right*.

Remarks

The template class stores a copy of a binary function object *Func* in `op`, and a copy of *right* in `value`. It defines its member function `operator()` as returning `op(left, value)`.

If `Func` is an object of type `Operation` and `c` is a constant, then `bind2nd (Func, c)` is equivalent to the `binder2nd` class constructor `binder2nd < Operation > (Func, c)` and more convenient.

Example

```

// functional_binder2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
        binder2nd<greater<int> >(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;

    // Compare using binder1st fixing 1st argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder1st<greater<int> >(greater<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
        << result2 << "." << endl;
}
/* Output:
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

const_mem_fun_ref_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a **const** member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type>
class const_mem_fun_ref_t
: public unary_function<Type, Result>
{
    explicit const_mem_fun_t(Result (Type::* Pm)() const);
    Result operator()(const Type& left) const;
};
```

Parameters

Pm

A pointer to the member function of class `Type` to be converted to a function object.

left

The object that the *Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(left.*Pm)() const`.

Example

The constructor of `const_mem_fun_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See [mem_fun_ref](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

const_mem_fun_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type>
class const_mem_fun_t : public unary_function <Type *, Result>
{
    explicit const_mem_fun_t(Result (Type::* Pm)() const);
    Result operator()(const Type* Pleft) const;
};
```

Parameters

Pm

A pointer to the member function of class `Type` to be converted to a function object.

Pleft

The object that the *Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(Pleft -> * Pm)() const`.

Example

The constructor of `const_mem_fun_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See [mem_fun](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)
[C++ Standard Library Reference](#)

const_mem_fun1_ref_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a **const** member function that takes a single argument to be called as a binary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type, class Arg>
class const_mem_fun1_ref_t
: public binary_function<Type, Arg, Result>
{
    explicit const_mem_fun1_ref_t(Result (Type::* Pm)(Arg) const);
    Result operator()(const Type& left, Arg right) const;
};
```

Parameters

Pm

A pointer to the member function of class `Type` to be converted to a function object.

left

The **const** object that the *Pm* member function is called on.

right

The argument that is being given to *Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(left.* Pm)(right) const`.

Example

The constructor of `const_mem_fun1_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See [mem_fun_ref](#) for examples of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

const_mem_fun1_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a **const** member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type, class Arg>
class const_mem_fun1_t : public binary_function<const Type *, Arg, Result>
{
    explicit const_mem_fun1_t(Result (Type::* member_ptr)(Arg) const);
    Result operator()(const Type* left, Arg right) const;
};
```

Parameters

member_ptr

A pointer to the member function of class `Type` to be converted to a function object.

left

The **const** object that the *member_ptr* member function is called on.

right

The argument that is being given to *member_ptr*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *member_ptr*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(left->member_ptr)(right) const`.

Example

The constructor of `const_mem_fun1_t` is rarely used directly. `mem_fn` is used to adapt member functions. See [mem_fn](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

divides Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the division operation (`operator/`) on its arguments.

Syntax

```
template <class Type = void>
struct divides : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator/
template <>
struct divides<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left)* std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* A type that supports an `operator/` that takes operands of the specified or inferred types.

Left

The left operand of the division operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the division operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left / Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator/`.

Example

```

// functional_divides.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector <double> v1, v2, v3 (6);
    vector <double>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 7.0 * i );
    }

    int j;
    for ( j = 1 ; j <= 6 ; j++ )
    {
        v2.push_back( 2.0 * j);
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise quotients of the elements of v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
        divides<double>( ) );

    cout << "The element-wise quotients are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}

/* Output:
The vector v1 = ( 0 7 14 21 28 35 )
The vector v2 = ( 2 4 6 8 10 12 )
The element-wise quotients are: ( 0 1.75 2.33333 2.625 2.8 2.91667 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

equal_to Struct

3/11/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the equality operation (`operator==`) on its arguments.

Syntax

```
template <class Type = void>
struct equal_to : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator==
template <>
struct equal_to<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) == std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator==` that takes operands of the specified or inferred types.

Left

The left operand of the equality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the equality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left == Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator==`.

Remarks

The objects of type *Type* must be equality-comparable. This requires that the `operator==` defined on the set of objects satisfies the mathematical properties of an equivalence relation. All of the built-in numeric and pointer types satisfy this requirement.

Example

```

// functional_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector <double> v1, v2, v3 ( 6 );
    vector <double>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 0 ; i <= 5 ; i+=2 )
    {
        v1.push_back( 2.0 *i );
        v1.push_back( 2.0 * i + 1.0 );
    }

    int j;
    for ( j = 0 ; j <= 5 ; j+=2 )
    {
        v2.push_back( - 2.0 * j );
        v2.push_back( 2.0 * j + 1.0 );
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Testing for the element-wise equality between v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        equal_to<double>( ) );

    cout << "The result of the element-wise equal_to comparison\n"
        << "between v1 & v2 is: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}

```

```

The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise equal_to comparison
between v1 & v2 is: ( 1 1 0 1 0 1 )

```

function Class

11/9/2018 • 9 minutes to read • [Edit Online](#)

Wrapper for a callable object.

Syntax

```
template <class Fty>
class function // Fty of type Ret(T1, T2, ..., TN)
    : public unary_function<T1, Ret>          // when Fty is Ret(T1)
    : public binary_function<T1, T2, Ret>     // when Fty is Ret(T1, T2)
{
public:
    typedef Ret result_type;

    function();
    function(nullptr_t);
    function(const function& right);
    template <class Fty2>
        function(Fty2 fn);
    template <class Fty2, class Alloc>
        function(reference_wrapper<Fty2>, const Alloc& Ax);

    template <class Fty2, class Alloc>
        void assign(Fty2, const Alloc& Ax);
    template <class Fty2, class Alloc>
        void assign(reference_wrapper<Fty2>, const Alloc& Ax);
    function& operator=(nullptr_t);
    function& operator=(const function&);
    template <class Fty2>
        function& operator=(Fty2);
    template <class Fty2>
        function& operator=(reference_wrapper<Fty2>);

    void swap(function&);
    explicit operator bool() const;

    result_type operator()(T1, T2, ....., TN) const;
    const std::type_info& target_type() const;
    template <class Fty2>
        Fty2 *target();

    template <class Fty2>
        const Fty2 *target() const;

    template <class Fty2>
        void operator==(const Fty2&) const = delete;
    template <class Fty2>
        void operator!=(const Fty2&) const = delete;
};
```

Parameters

Fty

The function type to wrap.

Ax

The allocator function.

Remarks

The template class is a call wrapper whose call signature is `Ret(T1, T2, ..., TN)`. You use it to enclose a variety of callable objects in a uniform wrapper.

Some member functions take an operand that names the desired target object. You can specify such an operand in several ways:

`fn` -- the callable object `fn`; after the call the `function` object holds a copy of `fn`

`fnref` -- the callable object named by `fnref.get()`; after the call the `function` object holds a reference to `fnref.get()`

`right` -- the callable object, if any, held by the `function` object `right`

`npc` -- a null pointer; after the call the `function` object is empty

In all cases, `INVOKE(f, t1, t2, ..., tN)`, where `f` is the callable object and `t1, t2, ..., tN` are lvalues of types `T1, T2, ..., TN` respectively, must be well-formed and, if `Ret` is not void, convertible to `Ret`.

An empty `function` object does not hold a callable object or a reference to a callable object.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>function</code>	Constructs a wrapper that either is empty or stores a callable object of arbitrary type with a fixed signature.

Typedefs

TYPE NAME	DESCRIPTION
<code>result_type</code>	The return type of the stored callable object.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Assigns a callable object to this function object.
<code>swap</code>	Swap two callable objects.
<code>target</code>	Tests if stored callable object is callable as specified.
<code>target_type</code>	Gets type information on the callable object.

Operators

OPERATOR	DESCRIPTION
<code>function::operator unspecified</code>	Tests if stored callable object exists.
<code>function::operator()</code>	Calls a callable object.
<code>function::operator=</code>	Replaces the stored callable object.

Requirements

Header: <functional>

Namespace: std

function::assign

Assigns a callable object to this function object.

```
template <class Fx, class Alloc>
void assign(
    Fx _Func,
    const Alloc& Ax);

template <class Fx, class Alloc>
void assign(
    reference_wrapper<Fx> _Fnref,
    const Alloc& Ax);
```

Parameters

_Func

A callable object.

_Fnref

A reference wrapper that contains a callable object.

Ax

An allocator object.

Remarks

The member functions each replace the `callable object` held by `*this` with the callable object passed as the `operand`. Both allocate storage with the allocator object *Ax*.

function::function

Constructs a wrapper that either is empty or stores a callable object of arbitrary type with a fixed signature.

```
function();
function(nullptr_t npc);
function(const function& right);
template <class Fx>
    function(Fx _Func);
template <class Fx>
    function(reference_wrapper<Fx> _Fnref);
template <class Fx, class Alloc>
    function(
        Fx _Func,
        const Alloc& Ax);

template <class Fx, class Alloc>
    function(
        reference_wrapper<Fx> _Fnref,
        const Alloc& Ax);
```

Parameters

right

The function object to copy.

Fx

The type of the callable object.

_Func

The callable object to wrap.

Alloc

The allocator type.

Ax

The allocator.

_Fnref

The callable object reference to wrap.

Remarks

The first two constructors construct an empty `function` object. The next three constructors construct a `function` object that holds the callable object passed as the operand. The last two constructors allocate storage with the allocator object `Ax`.

Example


```

// std_functional_function_function.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>
#include <vector>

int square(int val)
{
    return val * val;
}

class multiply_by
{
public:
    explicit multiply_by(const int n) : m_n(n) { }

    int operator()(const int x) const
    {
        return m_n * x;
    }

private:
    int m_n;
};

int main()
{
    typedef std::vector< std::function<int (int)> > vf_t;

    vf_t v;
    v.push_back(square);
    v.push_back(std::negate<int>());
    v.push_back(multiply_by(3));

    for (vf_t::const_iterator i = v.begin(); i != v.end(); ++i)
    {
        std::cout << (*i)(10) << std::endl;
    }

    std::function<int (int)> f = v[0];
    std::function<int (int)> g;

    if (f) {
        std::cout << "f is non-empty (correct)." << std::endl;
    } else {
        std::cout << "f is empty (can't happen)." << std::endl;
    }

    if (g) {
        std::cout << "g is non-empty (can't happen)." << std::endl;
    } else {
        std::cout << "g is empty (correct)." << std::endl;
    }

    return 0;
}

```

```

100
-10
30
f is non-empty (correct).
g is empty (correct).

```

function::operator unspecified

Tests if stored callable object exists.

```
operator unspecified();
```

Remarks

The operator returns a value that is convertible to **bool** with a true value only if the object is not empty. You use it to test whether the object is empty.

Example

```
// std_functional_function_operator_bool.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0;
    std::cout << std::boolalpha << "not empty == " << (bool)fn0 << std::endl;

    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "not empty == " << (bool)fn1 << std::endl;

    return (0);
}
```

```
not empty == false
not empty == true
```

function::operator()

Calls a callable object.

```
result_type operator()(
    T1 t1,
    T2 t2, ...,
    TN tN);
```

Parameters

TN

The type of the Nth call argument.

tN

The Nth call argument.

Remarks

The member function returns `INVOKE(fn, t1, t2, ..., tN, Ret)`, where `fn` is the target object stored in `*this`. You use it to call the wrapped callable object.

Example

```
// std_functional_function_operator_call.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
}
```

```
empty == false
val == -3
```

function::operator=

Replaces the stored callable object.

```
function& operator=(null_ptr_type npc);
function& operator=(const function& right);
template <class Fty>
    function& operator=(Fty fn);
template <class Fty>
    function& operator=(reference_wrapper<Fty> fnref);
```

Parameters

npc

A null pointer constant.

right

The function object to copy.

fn

The callable object to wrap.

fnref

The callable object reference to wrap.

Remarks

The operators each replace the callable object held by `*this` with the callable object passed as the operand.

Example

```
// std_functional_function_operator_as.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    fn1 = 0;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;

    fn1 = neg;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    fn1 = fn0;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    fn1 = std::cref(fn1);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
}
```

```
empty == false
val == -3
empty == true
empty == false
val == -3
empty == false
val == -3
empty == false
val == -3
```

function::result_type

The return type of the stored callable object.

```
typedef Ret result_type;
```

Remarks

The typedef is a synonym for the type `Ret` in the template's call signature. You use it to determine the return type of the wrapped callable object.

Example

```
// std_functional_function_result_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;

    std::function<int (int)>::result_type val = fn1(3);
    std::cout << "val == " << val << std::endl;

    return (0);
}
```

```
empty == false
val == -3
```

function::swap

Swap two callable objects.

```
void swap(function& right);
```

Parameters

right

The function object to swap with.

Remarks

The member function swaps the target objects between `*this` and *right*. It does so in constant time and throws no exceptions.

Example

```
// std_functional_function_swap.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << std::endl;

    fn0.swap(fn1);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
}
```

```
empty == false
val == -3
empty == true

empty == true
empty == false
val == -3
```

function::target

Tests if stored callable object is callable as specified.

```
template <class Fty2>
    Fty2 *target();
template <class Fty2>
    const Fty2 *target() const;
```

Parameters

Fty2

The target callable object type to test.

Remarks

The type *Fty2* must be callable for the argument types `T1, T2, ..., TN` and the return type `Ret`. If `target_type() == typeid(Fty2)`, the member template function returns the address of the target object; otherwise, it returns 0.

A type *Fty2* is callable for the argument types `T1, T2, ..., TN` and the return type `Ret` if, for lvalues `fn, t1, t2, ..., tN` of types `Fty2, T1, T2, ..., TN`, respectively, `INVOKE(fn, t1, t2, ..., tN)` is well-formed and, if `Ret` is not **void**, convertible to `Ret`.

Example

```

// std__functional__function_target.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    typedef int (*Myfun)(int);
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "no target == " << (fn0.target<Myfun>() == 0) << std::endl;

    Myfun *fptr = fn0.target<Myfun>();
    std::cout << "val == " << (*fptr)(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "no target == " << (fn1.target<Myfun>() == 0) << std::endl;

    return (0);
}

```

```

empty == false
no target == false
val == -3
empty == true
no target == true

```

function::target_type

Gets type information on the callable object.

```
const std::type_info& target_type() const;
```

Remarks

The member function returns `typeid(void)` if `*this` is empty, otherwise it returns `typeid(T)`, where `T` is the type of the target object.

Example

```

// std__functional__function_target_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "type == " << fn0.target_type().name() << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "type == " << fn1.target_type().name() << std::endl;

    return (0);
}

```

```

empty == false
type == int (__cdecl*)(int)
empty == true
type == void

```

See also

[mem_fn](#)

[reference_wrapper Class](#)

greater Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the greater-than operation (`operator>`) on its arguments.

Syntax

```
template <class Type = void>
struct greater : public binary_function <Type, Type, bool>
{
    bool operator()(
        const Type& Left,
        const Type& Right) const;

};

// specialized transparent functor for operator>
template <>
struct greater<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) > std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator>` that takes operands of the specified or inferred types.

Left

The left operand of the greater-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the greater-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left > Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator>`.

Remarks

The binary predicate `greater < Type >` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

```

// functional_greater.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i < 8 ; i++ )
    {
        v1.push_back( rand( ) );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    // use default binary predicate less<int>( )
    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order,
    // specify binary predicate greater<int>( )
    sort( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

```

Original vector v1 = (41 18467 6334 26500 19169 15724 11478 29358)
Sorted vector v1 = (41 6334 11478 15724 18467 19169 26500 29358)
Resorted vector v1 = (29358 26500 19169 18467 15724 11478 6334 41)

```

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

greater_equal Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the greater-than-or-equal-to operation (`operator>=`) on its arguments.

Syntax

```
template <class Type = void>
struct greater_equal : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator>=
template <>
struct greater_equal<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left)>= std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator>=` that takes operands of the specified or inferred types.

Left

The left operand of the greater-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the greater-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left >= Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator>=`.

Remarks

The binary predicate `greater_equal < Type >` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

```

// functional_greater_equal.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    v1.push_back( 6262 );
    v1.push_back( 6262 );
    for ( i = 0 ; i < 5 ; i++ )
    {
        v1.push_back( rand( ) );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    // use default binary predicate less<int>( )
    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order,
    // specify binary predicate greater_equal<int>( )
    sort( v1.begin( ), v1.end( ), greater_equal<int>( ) );
    cout << "Resorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

```

Original vector v1 = (6262 6262 41 18467 6334 26500 19169)
Sorted vector v1 = (41 6262 6262 6334 18467 19169 26500)
Resorted vector v1 = (26500 19169 18467 6334 6262 6262 41)

```

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

hash Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Computes hash code for a value.

Syntax

```
template <class Ty>
struct hash {
    size_t operator()(Ty val) const;
};
```

Remarks

The function object defines a hash function, suitable for mapping values of type *Ty* to a distribution of index values.

The member `operator()` returns a hash code for *val*, suitable for use with template classes `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. The standard library provides specializations for basic types: *Ty* may be any scalar type, including pointer types and enumeration types. In addition, there are specializations for the library types `string`, `wstring`, `u16string`, `u32string`, `string_view`, `wstring_view`, `u16string_view`, `u32string_view`, `bitset`, `error_code`, `error_condition`, `optional`, `shared_ptr`, `thread`, `type_index`, `unique_ptr`, `variant`, and `vector<bool>`.

Example

```
// std_functional_hash.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>
#include <unordered_set>

int main()
{
    std::unordered_set<int, std::hash<int> > c0;
    c0.insert(3);
    std::cout << *c0.find(3) << std::endl;

    return (0);
}
```

3

Requirements

Header: `<functional>`

Namespace: `std`

See also

[<unordered_map>](#)

[unordered_multimap Class](#)

[unordered_multiset Class](#)

[<unordered_set>](#)

is_bind_expression Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type generated by calling `bind`.

Syntax

```
template struct is_bind_expression { static const bool value; };
```

Remarks

The constant member `value` is true if the type `Ty` is a type returned by a call to `bind`, otherwise false.

Example

```
// std__functional__is_bind_expression.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

void square(double x)
{
    std::cout << x << "^2 == " << x * x << std::endl;
}

template<class Expr>
void test_for_bind(const Expr&)
{
    std::cout << std::is_bind_expression<Expr>::value << std::endl;
}

int main()
{
    test_for_bind(3.0 * 3.0);
    test_for_bind(std::bind(square, 3));

    return (0);
}
```

```
0
1
```

Requirements

Header: <functional>

Namespace: std

See also

[bind](#)

is_placeholder Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a placeholder.

Syntax

```
struct is_placeholder { static const int value; };
```

Remarks

The constant value `value` is 0 if the type `Ty` is not a placeholder; otherwise, its value is the position of the function call argument that it binds to. You use it to determine the value `N` for the Nth placeholder `_N`.

Example

```
// std__functional__is_placeholder.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

using namespace std::placeholders;

template<class Expr>
void test_for_placeholder(const Expr&)
{
    std::cout << std::is_placeholder<Expr>::value << std::endl;
}

int main()
{
    test_for_placeholder(3.0);
    test_for_placeholder(_3);

    return (0);
}
```

```
0
3
```

Requirements

Header: <functional>

Namespace: std

See also

[_1 Object](#)

less Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the less-than operation (`operator<`) on its arguments.

Syntax

```
template <class Type = void>
struct less : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator<
template <>
struct less<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) < std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator<` that takes operands of the specified or inferred types.

Left

The left operand of the less-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the less-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left < Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator<`.

Remarks

The binary predicate `less < Type >` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

```

// functional_less.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

struct MyStruct {
    MyStruct(int i) : m_i(i){}

    bool operator < (const MyStruct & rhs) const {
        return m_i < rhs.m_i;
    }

    int m_i;
};

int main() {
    using namespace std;
    vector <MyStruct> v1;
    vector <MyStruct>::iterator Iter1;
    vector <MyStruct>::reverse_iterator rIter1;

    int i;
    for ( i = 0 ; i < 7 ; i++ )
        v1.push_back( MyStruct(rand()));

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
        cout << Iter1->m_i << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    sort( v1.begin( ), v1.end( ), less<MyStruct>());

    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
        cout << Iter1->m_i << " ";
    cout << ")" << endl;
}

```

Output

```

Original vector v1 = (41 18467 6334 26500 19169 15724 11478)
Sorted vector v1 = (41 6334 11478 15724 18467 19169 26500)

```

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

less_equal Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the less-than-or-equal-to operation (`operator<=`) on its arguments.

Syntax

```
template <class Type = void>
struct less_equal : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator<=
template <>
struct less_equal<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) <= std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator<=` that takes operands of the specified or inferred types.

Left

The left operand of the less-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the less-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left <= Right`. The specialized template does perfect forwarding of the result, which has the type returned by `operator<=`.

Remarks

The binary predicate `less_equal < Type >` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

```

// functional_less_equal.cpp
// compile with: /EHsc
#define _CRT_RAND_S
#include <stdlib.h>
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;
    vector<int>::reverse_iterator rIter1;
    unsigned int randomNumber;

    int i;
    for ( i = 0 ; i < 5 ; i++ )
    {
        if ( rand_s( &randomNumber ) == 0 )
        {
            // Convert the random number to be between 1 - 50000
            // This is done for readability purposes
            randomNumber = ( unsigned int) ((double)randomNumber /
                (double) UINT_MAX * 50000) + 1;

            v1.push_back( randomNumber );
        }
    }
    for ( i = 0 ; i < 3 ; i++ )
    {
        v1.push_back( 2836 );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in ascending order,
    // use the binary predicate less_equal<int>( )
    sort( v1.begin( ), v1.end( ), less_equal<int>( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}

```

Sample Output

```

Original vector v1 = (31247 37154 48755 15251 6205 2836 2836 2836)
Sorted vector v1 = (2836 2836 2836 6205 15251 31247 37154 48755)

```

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

logical_and Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the logical conjunction operation (`operator&&`) on its arguments.

Syntax

```
template <class Type = void>
struct logical_and : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator&&
template <>
struct logical_and<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) && std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator&&` that takes operands of the specified or inferred types.

Left

The left operand of the logical conjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the logical conjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left && Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator&&`.

Remarks

For user-defined types, there is no short-circuiting of operand evaluation. Both arguments are evaluated by `operator&&`.

Example

```
// functional_logical_and.cpp
// compile with: /EHsc

#define _CRT_RAND_S
#include <stdlib.h>
#include <deque>
```

```

#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;
    deque<bool> d1, d2, d3( 7 );
    deque<bool>::iterator iter1, iter2, iter3;

    unsigned int randomValue;

    int i;
    for ( i = 0 ; i < 7 ; i++ )
    {
        if ( rand_s( &randomValue ) == 0 )
        {
            d1.push_back((bool)(( randomValue % 2 ) != 0));
        }
    }

    int j;
    for ( j = 0 ; j < 7 ; j++ )
    {
        if ( rand_s( &randomValue ) == 0 )
        {
            d2.push_back((bool)(( randomValue % 2 ) != 0));
        }
    }

    cout << boolalpha;    // boolalpha I/O flag on

    cout << "Original deque:\n d1 = ( " ;
    for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
        cout << *iter1 << " ";
    cout << ")" << endl;

    cout << "Original deque:\n d2 = ( " ;
    for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
        cout << *iter2 << " ";
    cout << ")" << endl;

    // To find element-wise conjunction of the truth values
    // of d1 & d2, use the logical_and function object
    transform( d1.begin( ), d1.end( ), d2.begin( ),
        d3.begin( ), logical_and<bool>( ) );
    cout << "The deque which is the conjunction of d1 & d2 is:\n d3 = ( " ;
    for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
        cout << *iter3 << " ";
    cout << ")" << endl;
}

/* Output:
Original deque:
d1 = ( true true true true true false false )
Original deque:
d2 = ( true false true true false true false )
The deque which is the conjunction of d1 & d2 is:
d3 = ( true false true true false false false )
*/

```

Requirements

Header: <functional>

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

logical_not Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the logical not operation (`operator!`) on its argument.

Syntax

```
template <class Type = void>
struct logical_not : public unary_function<Type, bool>
{
    bool operator()(const Type& Left) const;
};

// specialized transparent functor for operator!
template <>
struct logical_not<void>
{
    template <class Type>
    auto operator()(Type&& Left) const`
        -> decltype(!std::forward<Type>(Left));
};
```

Parameters

Type

Any type that supports an `operator!` that takes an operand of the specified or inferred type.

Left

The operand of the logical not operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *Type*.

Return Value

The result of `!Left`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator!`.

Example

```

// functional_logical_not.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;
    deque<bool> d1, d2 ( 7 );
    deque<bool>::iterator iter1, iter2;

    int i;
    for ( i = 0 ; i < 7 ; i++ )
    {
        d1.push_back((bool)((i % 2) != 0));
    }

    cout << boolalpha;    // boolalpha I/O flag on

    cout << "Original deque:\n d1 = ( " ;
    for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
        cout << *iter1 << " ";
    cout << ")" << endl;

    // To flip all the truth values of the elements,
    // use the logical_not function object
    transform( d1.begin( ), d1.end( ), d2.begin( ), logical_not<bool>( ) );
    cout << "The deque with its values negated is:\n d2 = ( " ;
    for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
        cout << *iter2 << " ";
    cout << ")" << endl;
}
/* Output:
Original deque:
d1 = ( false true false true false true false )
The deque with its values negated is:
d2 = ( true false true false true false true )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

logical_or Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the logical disjunction operation (`operator||`) on its arguments.

Syntax

```
template <class Type = void>
struct logical_or : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator||
template <>
struct logical_or<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) || std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator||` that takes operands of the specified or inferred types.

Left

The left operand of the logical disjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the logical disjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left || Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator||`.

Remarks

For user-defined types, there is no short-circuiting of operand evaluation. Both arguments are evaluated by `operator||`.

Example

```

// functional_logical_or.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;
    deque<bool> d1, d2, d3( 7 );
    deque<bool>::iterator iter1, iter2, iter3;

    int i;
    for ( i = 0 ; i < 7 ; i++ )
    {
        d1.push_back((bool)((rand() % 2) != 0));
    }

    int j;
    for ( j = 0 ; j < 7 ; j++ )
    {
        d2.push_back((bool)((rand() % 2) != 0));
    }

    cout << boolalpha;    // boolalpha I/O flag on

    cout << "Original deque:\n d1 = ( " ;
    for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
        cout << *iter1 << " ";
    cout << ")" << endl;

    cout << "Original deque:\n d2 = ( " ;
    for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
        cout << *iter2 << " ";
    cout << ")" << endl;

    // To find element-wise disjunction of the truth values
    // of d1 & d2, use the logical_or function object
    transform( d1.begin( ), d1.end( ), d2.begin( ),
        d3.begin( ), logical_or<bool>( ) );
    cout << "The deque which is the disjunction of d1 & d2 is:\n d3 = ( " ;
    for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
        cout << *iter3 << " ";
    cout << ")" << endl;
}
/* Output:
Original deque:
d1 = ( true true false false true false false )
Original deque:
d2 = ( false false false true true true true )
The deque which is the disjunction of d1 & d2 is:
d3 = ( true true false true true true true )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

mem_fun_ref_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type>
class mem_fun_ref_t : public unary_function<Type, Result> {
    explicit mem_fun_ref_t(
        Result (Type::* _Pm)());

    Result operator()(Type& left) const;
};
```

Parameters

_Pm

A pointer to the member function of class `Type` to be converted to a function object.

left

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(left.*_Pm)()`.

Example

The constructor of `mem_fun_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See [mem_fun_ref](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

mem_fun_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type>
class mem_fun_t : public unary_function<Type *, Result> {
    explicit mem_fun_t(Result (Type::* _Pm)());

    Result operator()(Type* _Pleft) const;
};
```

Parameters

_Pm

A pointer to the member function of class `Type` to be converted to a function object.

_Pleft

The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(_Pleft -> * _Pm)()`.

Example

The constructor of `mem_fun_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See [mem_fun](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[<functional>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

mem_fun1_ref_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type, class Arg>
class mem_fun1_ref_t : public binary_function<Type, Arg, Result> {
    explicit mem_fun1_ref_t(
        Result (Type::* _Pm)(Arg));

    Result operator()(
        Type& left,
        Arg right) const;

};
```

Parameters

_Pm

A pointer to the member function of class `Type` to be converted to a function object.

left

The object that the *_Pm* member function is called on.

right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (**left**.* *_Pm*)(**right**).

Example

The constructor of `mem_fun1_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See [mem_fun_ref](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

mem_fun1_t Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Result, class Type, class Arg>
class mem_fun1_t : public binary_function<Type *, Arg, Result> {
    explicit mem_fun1_t(
        Result (Type::* _Pm)(Arg));

    Result operator()(
        Type* _Pleft,
        Arg right) const;

};
```

Parameters

_Pm

A pointer to the member function of class `Type` to be converted to a function object.

_Pleft

The object that the *_Pm* member function is called on.

right

The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The template class stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(_Pleft->* _Pm)(right)`.

Example

The constructor of `mem_fun1_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See [mem_fun](#) for an example of how to use member function adaptors.

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

minus Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the subtraction operation (binary `operator-`) on its arguments.

Syntax

```
template <class Type = void>
struct minus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator-
template <>
struct minus<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) - std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* A type that supports a binary `operator-` that takes operands of the specified or inferred types.

Left

The left operand of the operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left - Right`. The specialized template does perfect forwarding of the result, which has the type returned by `operator-`.

Example

```

// functional_minus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector<int> v1, v2, v3 ( 6 );
    vector<int>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 4 * i + 1);
    }

    int j;
    for ( j = 0 ; j <= 5 ; j++ )
    {
        v2.push_back( 3 * j - 1);
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise difference of the elements of v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        minus<int>( ) );

    cout << "The element-wise differences between v1 and v2 are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 1 5 9 13 17 21 )
The vector v2 = ( -1 2 5 8 11 14 )
The element-wise differences between v1 and v2 are: ( 2 3 4 5 6 7 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)
[C++ Standard Library Reference](#)

modulus Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the modulus division operation (`operator%`) on its arguments.

Syntax

```
template <class Type = void>
struct modulus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator%
template <>
struct modulus<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) % std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator%` that takes operands of the specified or inferred types.

Left

The left operand of the modulus operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the modulus operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left % Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator%`.

Remarks

The `modulus` functor is restricted to integral types for the basic data types, or to user-defined types that implement `operator%`.

Example

```

// functional_modulus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector<int> v1, v2, v3 ( 6 );
    vector<int>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 1 ; i <= 6 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int j;
    for ( j = 1 ; j <= 6 ; j++ )
    {
        v2.push_back( 3 * j );
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise remainders of the elements of v1 & v2
    transform (v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        modulus<int>() );

    cout << "The element-wise remainders of the modular division\n are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 5 10 15 20 25 30 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise remainders of the modular division
are: ( 2 4 6 8 10 12 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

multiplies Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the multiplication operation (binary `operator*`) on its arguments.

Syntax

```
template <class Type = void>
struct multiplies : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator*
template <>
struct multiplies<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) * std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* A type that supports a binary `operator*` that takes operands of the specified or inferred types.

Left

The left operand of the multiplication operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the multiplication operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left * Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator*`.

Example


```

// functional_multiplies.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector<int> v1, v2, v3 ( 6 );
    vector<int>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 1 ; i <= 6 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    int j;
    for ( j = 1 ; j <= 6 ; j++ )
    {
        v2.push_back( 3 * j );
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise products of the elements of v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        multiplies<int>( ) );

    cout << "The element-wise products of vectors V1 & v2\n are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 2 4 6 8 10 12 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise products of vectors V1 & v2
are: ( 6 24 54 96 150 216 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

negate Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the arithmetic negation operation (unary `operator-`) on its argument.

Syntax

```
template <class Type = void>
struct negate : public unary_function<Type, Type>
{
    Type operator()(const Type& Left) const;
};

// specialized transparent functor for unary operator-
template <>
struct negate<void>
{
    template <class Type>
    auto operator()(Type&& Left) const`
        -> decltype(-std::forward<Type>(Left));
};
```

Parameters

Type

Any type that supports an `operator-` that takes an operand of the specified or inferred type.

Left

The operand to be negated. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *Type*.

Return Value

The result of `-Left`. The specialized template does perfect forwarding of the result, which has the type that's returned by unary `operator-`.

Example

```

// functional_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector<int> v1, v2 ( 8 );
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = -2 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Finding the element-wise negatives of the vector v1
    transform ( v1.begin( ), v1.end( ), v2.begin( ), negate<int>( ) );

    cout << "The negated elements of the vector = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( -10 -5 0 5 10 15 20 25 )
The negated elements of the vector = ( 10 5 0 -5 -10 -15 -20 -25 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

not_equal_to Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A binary predicate that performs the inequality operation (`operator!=`) on its arguments.

Syntax

```
template <class Type = void>
struct not_equal_to : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator!=
template <>
struct not_equal_to<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) != std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* Any type that supports an `operator!=` that takes operands of the specified or inferred types.

Left

The left operand of the inequality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the inequality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left != Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator!=`.

Remarks

The objects of type *Type* must be equality-comparable. This requires that the `operator!=` defined on the set of objects satisfies the mathematical properties of an equivalence relation. All of the built-in numeric and pointer types satisfy this requirement.

Example

```

// functional_not_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector <double> v1, v2, v3 (6);
    vector <double>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 0 ; i <= 5 ; i+=2 )
    {
        v1.push_back( 2.0 *i );
        v1.push_back( 2.0 * i + 1.0 );
    }

    int j;
    for ( j = 0 ; j <= 5 ; j+=2 )
    {
        v2.push_back( - 2.0 * j );
        v2.push_back( 2.0 * j + 1.0 );
    }

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Testing for the element-wise equality between v1 & v2
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
        not_equal_to<double>( ) );

    cout << "The result of the element-wise not_equal_to comparsion\n"
        << "between v1 & v2 is: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise not_equal_to comparsion
between v1 & v2 is: ( 0 0 1 0 1 0 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

plus Struct

2/28/2019 • 2 minutes to read • [Edit Online](#)

A predefined function object that performs the addition operation (binary `operator+`) on its arguments.

Syntax

```
template <class Type = void>
struct plus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator+
template <>
struct plus<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const`
        -> decltype(std::forward<T>(Left) + std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U* A type that supports a binary `operator+` that takes operands of the specified or inferred types.

Left

The left operand of the addition operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the addition operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left + Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by binary `operator+`.

Example

```

// functional_plus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
    vector <double> v1, v2, v3 ( 6 );
    vector <double>::iterator Iter1, Iter2, Iter3;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1.push_back( 4 * i );

    int j;
    for ( j = 0 ; j <= 5 ; j++ )
        v2.push_back( -2.0 * j - 4 );

    cout << "The vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "The vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Finding the element-wise sums of the elements of v1 & v2
    transform (v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ), plus<double>( ) );

    cout << "The element-wise sums are: ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
/* Output:
The vector v1 = ( 0 4 8 12 16 20 )
The vector v2 = ( -4 -6 -8 -10 -12 -14 )
The element-wise sums are: ( -4 -2 0 2 4 6 )
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

pointer_to_binary_function Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

Converts a binary function pointer into an adaptable binary function. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function
    : public binary_function <Arg1, Arg2, Result>
{
public:
    explicit pointer_to_binary_function(
        Result(*pfunc)(Arg1, Arg2));
    Result operator()(Arg1 left, Arg2 right) const;
};
```

Parameters

pfunc

The binary function to be converted.

left

The left object that the **pfunc* is called on.

right

The right object that the **pfunc* is called on.

Return Value

The template class stores a copy of `pfunc`. It defines its member function `operator()` as returning `(* pfunc)(Left, right)`.

Remarks

A binary function pointer is a function object and may be passed to any C++ Standard Library algorithm that is expecting a binary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types `first_argument_type`, `second_argument_type`, and `result_type` that make such an adaptation possible. The conversion by `pointer_to_binary_function` allows the function adaptors to work with binary function pointers.

Example

The constructor of `pointer_to_binary_function` is rarely used directly. See the helper function `ptr_fun` for an example of how to declare and use the `pointer_to_binary_function` adaptor predicate.

Requirements

Header: <functional>

Namespace: std

See also

[C++ Standard Library Reference](#)

pointer_to_unary_function Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

Converts a unary function pointer into an adaptable unary function. Deprecated in C++11, removed in C++17.

Syntax

```
template <class Arg, class Result>
class pointer_to_unary_function
    : public unary_function<Arg, Result>
{
public:
    explicit pointer_to_unary_function(Result(*pfunc)(Arg));
    Result operator()(Arg left) const;
};
```

Parameters

pfunc

The binary function to be converted.

left

The object that the **pfunc* is called on.

Return Value

The template class stores a copy of `pfunc`. It defines its member function `operator()` as returning `(* pfunc)(Left)`.

Remarks

A unary function pointer is a function object and may be passed to any C++ Standard Library algorithm that is expecting a unary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types `argument_type` and `result_type` that make such an adaptation possible. The conversion by `pointer_to_unary_function` allows the function adaptors to work with binary function pointers.

Example

The constructor of `pointer_to_unary_function` is rarely used directly. See the helper function `ptr_fun` for an example of how to declare and use the `pointer_to_unary_function` adaptor predicate.

Requirements

Header: `<functional>`

Namespace: `std`

See also

[C++ Standard Library Reference](#)

reference_wrapper Class

11/9/2018 • 3 minutes to read • [Edit Online](#)

Wraps a reference.

Syntax

```
template <class Ty>
class reference_wrapper
{
public:
    typedef Ty type;

    reference_wrapper(Ty&) noexcept;
    operator Ty&() const noexcept;
    Ty& get() const noexcept;

    template <class... Types>
    auto operator()(Types&&... args) const ->
        decltype(std::invoke(get(), std::forward<Types>(args)...));

private:
    Ty *ptr; // exposition only
};
```

Remarks

A `reference_wrapper<Ty>` is a copy constructible and copy assignable wrapper around a reference to an object or a function of type `Ty`, and holds a pointer that points to an object of that type. A `reference_wrapper` can be used to store references in standard containers, and to pass objects by reference to `std::bind`.

The type `Ty` must be an object type or a function type, or a static assert fails at compile time.

The helper functions `std::ref` and `std::cref` can be used to create `reference_wrapper` objects.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>reference_wrapper</code>	Constructs a <code>reference_wrapper</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>result_type</code>	The weak result type of the wrapped reference.
<code>type</code>	The type of the wrapped reference.

Member functions

MEMBER FUNCTION	DESCRIPTION
get	Obtains the wrapped reference.

Operators

OPERATOR	DESCRIPTION
reference_wrapper::operator Ty&	Gets a pointer to the wrapped reference.
reference_wrapper::operator()	Calls the wrapped reference.

Requirements

Header: <functional>

Namespace: std

reference_wrapper::get

Obtains the wrapped reference.

```
Ty& get() const noexcept;
```

Remarks

The member function returns the wrapped reference.

Example

```
// std__functional__reference_wrapper_get.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int main() {
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << rwi << std::endl;
    rwi.get() = -1;
    std::cout << "i = " << i << std::endl;

    return (0);
}
```

```
i = 1
rwi = 1
i = -1
```

reference_wrapper::operator Ty&

Gets the wrapped reference.

```
operator Ty&() const noexcept;
```

Remarks

The member operator returns `*ptr`.

Example

```
// std__functional__reference_wrapper_operator_cast.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int main() {
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "(int)rwi = " << (int)rwi << std::endl;

    return (0);
}
```

```
i = 1
(int)rwi = 1
```

reference_wrapper::operator()

Calls the wrapped reference.

```
template <class... Types>
auto operator()(Types&&... args);
```

Parameters

Types

The argument list types.

args

The argument list.

Remarks

The template member `operator()` returns `std::invoke(get(), std::forward<Types>(args)...) .`

Example

```
// std_functional_reference_wrapper_operator_call.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    std::reference_wrapper<int (int)> rwi(neg);

    std::cout << "rwi(3) = " << rwi(3) << std::endl;

    return (0);
}
```

```
rwi(3) = -3
```

reference_wrapper::reference_wrapper

Constructs a `reference_wrapper`.

```
reference_wrapper(Ty& val) noexcept;
```

Parameters

Ty

The type to wrap.

val

The value to wrap.

Remarks

The constructor sets the stored value `ptr` to `&val`.

Example

```
// std_functional_reference_wrapper_reference_wrapper.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << rwi << std::endl;
    rwi.get() = -1;
    std::cout << "i = " << i << std::endl;

    return (0);
}
```

```
i = 1
rwi = 1
i = -1
```

reference_wrapper::result_type

The weak result type of the wrapped reference.

```
typedef R result_type;
```

Remarks

The `result_type` typedef is a synonym for the weak result type of a wrapped function. This typedef is only meaningful for function types.

Example

```
// std_functional_reference_wrapper_result_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    typedef std::reference_wrapper<int (int)> Mywrapper;
    Mywrapper rwi(neg);
    Mywrapper::result_type val = rwi(3);

    std::cout << "val = " << val << std::endl;

    return (0);
}
```

```
val = -3
```

reference_wrapper::type

The type of the wrapped reference.

```
typedef Ty type;
```

Remarks

The typedef is a synonym for the template argument `Ty`.

Example

```
// std_functional_reference_wrapper_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    int i = 1;
    typedef std::reference_wrapper<int> Mywrapper;
    Mywrapper rwi(i);
    Mywrapper::type val = rwi.get();

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << val << std::endl;

    return (0);
}
```

```
i = 1
rwi = 1
```

See also

[cref](#)

[ref](#)

unary_function Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

An empty base struct that defines types that may be inherited by derived classes that provides a unary function object.

Syntax

```
struct unary_function
{
    typedef Arg argument_type;
    typedef Result result_type;
};
```

Remarks

The template struct serves as a base for classes that define a member function of the form **result_type** `operator()` (**const** **argument_type**&) **const**.

All such derived unary functions can refer to their sole argument type as **argument_type** and their return type as **result_type**.

Example

```

// functional_unary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan10: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 10);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greaterthan10());
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;
}

/* Output:
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
*/

```

Requirements

Header: <functional>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

unary_negate Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

A template class providing a member function that negates the return value of a specified unary function. Deprecated in C++17 in favor of [not_fn](#).

Syntax

```
template <class Predicate>
class unary_negate
    : public unaryFunction<typename Predicate::argument_type, bool>
{
public:
    explicit unary_negate(const Predicate& Func);
    bool operator()(const typename Predicate::argument_type& left) const;
};
```

Parameters

Func

The unary function to be negated.

left

The operand of the unary function to be negated.

Return Value

The negation of the unary function.

Remarks

The template class stores a copy of a unary function object *_Func*. It defines its member function `operator()` as returning `!_Func(left)`.

The constructor of `unary_negate` is rarely used directly. The helper function [not1](#) provides an easier way to declare and use the **unary_negator** adaptor predicate.

Example

```

// functional_unary_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
        << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        unary_negate<bindefun2nd <greater<int> > >(bind2nd(greater<int>(), 10))));

    // The following helper function not1 also works for the above line
    // not1(bind2nd(greater<int>(), 10));

    cout << "The number of elements in v1 not greater than 10 is: "
        << result2 << "." << endl;
}
/* Output:
The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.
*/

```

Requirements

Header: `<functional>`

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<future>

10/31/2018 • 3 minutes to read • [Edit Online](#)

Include the standard header <future> to define template classes and supporting templates that simplify running a function—possibly in a separate thread—and retrieving its result. The result is either the value that is returned by the function or an exception that is emitted by the function but is not caught in the function.

This header uses Concurrency Runtime (ConcRT) so that you can use it together with other ConcRT mechanisms. For more information about ConcRT, see [Concurrency Runtime](#).

Syntax

```
#include <future>
```

Remarks

NOTE

In code that is compiled by using **/clr**, this header is blocked.

An *asynchronous provider* stores the result of a function call. An *asynchronous return object* is used to retrieve the result of a function call. An *associated asynchronous state* provides communication between an asynchronous provider and one or more asynchronous return objects.

A program does not directly create any associated asynchronous state objects. The program creates an asynchronous provider whenever it needs one and from that it creates an asynchronous return object that shares its associated asynchronous state with the provider. Asynchronous providers and asynchronous return objects manage the objects that hold their shared associated asynchronous state. When the last object that references the associated asynchronous state releases it, the object that holds the associated asynchronous state is destroyed.

An asynchronous provider or an asynchronous return object that has no associated asynchronous state is *empty*.

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

The template function `async` and the template classes `promise` and `packaged_task` are asynchronous providers. The template classes `future` and `shared_future` describe asynchronous return objects.

Each of the template classes `promise`, `future`, and `shared_future` has a specialization for the type **void** and a partial specialization for storing and retrieving a value by reference. These specializations differ from the primary template only in the signatures and semantics of the functions that store and retrieve the returned value.

The template classes `future` and `shared_future` never block in their destructors, except in one case that's preserved for backward compatibility: Unlike all other futures, for a `future`—or the last `shared_future`—that's attached to a task started with `std::async`, the destructor blocks if the task has not completed; that is, it blocks if this thread did not yet call `.get()` or `.wait()` and the task is still running. The following usability note has been added to the description of `std::async` in the draft standard: "[Note: If a future obtained from `std::async` is moved outside the local scope, other code that uses the future must be aware that the future's destructor may block for the shared state to become ready.—end note]" In all other cases, `future` and `shared_future` destructors are required and are guaranteed to never block.

Members

Classes

NAME	DESCRIPTION
future Class	Describes an asynchronous return object.
future_error Class	Describes an exception object that can be thrown by methods of types that manage <code>future</code> objects.
packaged_task Class	Describes an asynchronous provider that is a call wrapper and whose call signature is <code>Ty(ArgTypes...)</code> . Its associated asynchronous state holds a copy of its callable object in addition to the potential result.
promise Class	Describes an asynchronous provider.
shared_future Class	Describes an asynchronous return object. In contrast with a <code>future</code> object, an asynchronous provider can be associated with any number of <code>shared_future</code> objects.

Structures

NAME	DESCRIPTION
is_error_code_enum Structure	Specialization that indicates that <code>future_errc</code> is suitable for storing an <code>error_code</code> .
uses_allocator Structure	Specialization that always holds true.

Functions

NAME	DESCRIPTION
async	Represents an asynchronous provider.
future_category	Returns a reference to the <code>error_category</code> object that characterizes errors that are associated with <code>future</code> objects.
make_error_code	Creates an <code>error_code</code> that has the <code>error_category</code> object that characterizes <code>future</code> errors.
make_error_condition	Creates an <code>error_condition</code> that has the <code>error_category</code> object that characterizes <code>future</code> errors.
swap	Exchanges the associated asynchronous state of one <code>promise</code> object with that of another.

Enumerations

NAME	DESCRIPTION
future_errc	Supplies symbolic names for the errors that are reported by the <code>future_error</code> class.
future_status	Supplies symbolic names for the reasons that a timed wait function can return.
launch	Represents a bitmask type that describes the possible modes for the template function <code>async</code> .

See also

[Header Files Reference](#)

<future> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

async	future_category	make_error_code
make_error_condition	swap	

async

Represents an *asynchronous provider*.

```
template <class Fn, class... ArgTypes>
future<typename result_of<Fn(ArgTypes...)>::type>
    async(Fn&& fn, ArgTypes&&... args);

template <class Fn, class... ArgTypes>
future<typename result_of<Fn(ArgTypes...)>::type>
    async(launch policy, Fn&& fn, ArgTypes&&... args);
```

Parameters

policy

A [launch](#) value.

Remarks

Definitions of abbreviations:

<i>dfn</i>	The result of calling <code>decay_copy(forward<Fn>(fn))</code> .
<i>dargs</i>	The results of the calls <code>decay_copy(forward<ArgTypes>(args...))</code> .
<i>Ty</i>	The type <code>result_of<Fn(ArgTypes...)>::type</code> .

The first template function returns `async(launch::any, fn, args...)`.

The second function returns a `future<Ty>` object whose *associated asynchronous state* holds a result together with the values of *dfn* and *dargs* and a thread object to manage a separate thread of execution.

Unless `decay<Fn>::type` is a type other than `launch`, the second function does not participate in overload resolution.

The C++ standard states that if `policy` is `launch::async`, the function creates a new thread. However the Microsoft implementation is currently non-conforming. It obtains its threads from the Windows ThreadPool, which in some cases may provide a recycled thread rather than a new one. This means that the `launch::async` policy is actually implemented as `launch::async|launch::deferred`. Another implication of the ThreadPool-based implementation is that there is no guarantee that thread-local variables will be destroyed when the thread completes. If the thread is recycled and provided to a new call to `async`, the old variables will still exist. We therefore recommend that you do

not use thread-local variables with `async`.

If *policy* is `launch::deferred`, the function marks its associated asynchronous state as holding a *deferred function* and returns. The first call to any non-timed function that waits for the associated asynchronous state to be ready in effect calls the deferred function by evaluating `INVOKE(dfn, dargs..., Ty)`.

In all cases, the associated asynchronous state of the `future` object is not set to *ready* until the evaluation of `INVOKE(dfn, dargs..., Ty)` completes, either by throwing an exception or by returning normally. The result of the associated asynchronous state is an exception if one was thrown, or any value that's returned by the evaluation.

NOTE

For a `future`—or the last `shared_future`—that's attached to a task started with `std::async`, the destructor blocks if the task has not completed; that is, it blocks if this thread did not yet call `.get()` or `.wait()` and the task is still running. If a `future` obtained from `std::async` is moved outside the local scope, other code that uses it must be aware that its destructor may block for the shared state to become ready.

The pseudo-function `INVOKE` is defined in [<functional>](#).

future_category

Returns a reference to the `error_category` object that characterizes errors that are associated with `future` objects.

```
const error_category& future_category() noexcept;
```

make_error_code

Creates an `error_code` together with the `error_category` object that characterizes `future` errors.

```
inline error_code make_error_code(future_errc Errno) noexcept;
```

Parameters

Errno

A `future_errc` value that identifies the reported error.

Return Value

```
error_code(static_cast<int>(Errno), future_category());
```

make_error_condition

Creates an `error_condition` together with the `error_category` object that characterizes `future` errors.

```
inline error_condition make_error_condition(future_errc Errno) noexcept;
```

Parameters

Errno

A `future_errc` value that identifies the reported error.

Return Value

```
error_condition(static_cast<int>(Errno), future_category());
```

swap

Exchanges the *associated asynchronous state* of one `promise` object with that of another.

```
template <class Ty>
void swap(promise<Ty>& Left, promise<Ty>& Right) noexcept;

template <class Ty, class... ArgTypes>
void swap(packaged_task<Ty(ArgTypes...)>& Left, packaged_task<Ty(ArgTypes...)>& Right) noexcept;
```

Parameters

Left

The left `promise` object.

Right

The right `promise` object.

See also

[<future>](#)

<future> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

future_errc	future_status	launch

future_errc Enumeration

Supplies symbolic names for all of the errors that are reported by the [future_error](#) class.

```
class future_errc {
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
};
```

future_status Enumeration

Supplies symbolic names for the reasons that a timed wait function can return.

```
enum future_status{
    ready,
    timeout,
    deferred
};
```

launch Enumeration

Represents a bitmask type that describes the possible modes for the template function [async](#).

```
class launch{
    async,
    deferred
};
```

See also

[<future>](#)

future Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an *asynchronous return object*.

Syntax

```
template <class Ty>
class future;
```

Remarks

Each standard *asynchronous provider* returns an object whose type is an instantiation of this template. A `future` object provides the only access to the asynchronous provider that it is associated with. If you need multiple asynchronous return objects that are associated with the same asynchronous provider, copy the `future` object to a `shared_future` object.

Members

Public Constructors

NAME	DESCRIPTION
<code>future</code>	Constructs a <code>future</code> object.

Public Methods

NAME	DESCRIPTION
<code>get</code>	Retrieves the result that is stored in the associated asynchronous state.
<code>share</code>	Converts the object to a <code>shared_future</code> .
<code>valid</code>	Specifies whether the object is not empty.
<code>wait</code>	Blocks the current thread until the associated asynchronous state is ready.
<code>wait_for</code>	Blocks until the associated asynchronous state is ready or until the specified time has elapsed.
<code>wait_until</code>	Blocks until the associated asynchronous state is ready or until a specified point in time.

Public Operators

NAME	DESCRIPTION
<code>future::operator=</code>	Transfers the associated asynchronous state from a specified object.

Requirements

Header: `<future>`

Namespace: `std`

future::future Constructor

Constructs a `future` object.

```
future() noexcept;
future(future&& Other) noexcept;
```

Parameters

Other

A `future` object.

Remarks

The first constructor constructs a `future` object that has no associated asynchronous state.

The second constructor constructs a `future` object and transfers the associated asynchronous state from *Other*. *Other* no longer has an associated asynchronous state.

future::get

Retrieves the result that is stored in the associated asynchronous state.

```
Ty get();
```

Return Value

If the result is an exception, the method rethrows it. Otherwise, the result is returned.

Remarks

Before it retrieves the result, this method blocks the current thread until the associated asynchronous state is ready.

For the partial specialization `future<Ty&>`, the stored value is effectively a reference to the object that was passed to the asynchronous provider as the return value.

Because no stored value exists for the specialization `future<void>`, the method returns **void**.

In other specializations, the method moves its return value from the stored value. Therefore, call this method only once.

future::operator=

Transfers an associated asynchronous state from a specified object.

```
future& operator=(future&& Right) noexcept;
```

Parameters

Right

A `future` object.

Return Value

`*this`

Remarks

After the transfer, *Right* no longer has an associated asynchronous state.

future::share

Converts the object to a [shared_future](#) object.

```
shared_future<Ty> share();
```

Return Value

`shared_future(move(*this))`

future::valid

Specifies whether the object has an associated asynchronous state.

```
bool valid() noexcept;
```

Return Value

true if the object has an associated asynchronous state; otherwise, **false**.

future::wait

Blocks the current thread until the associated asynchronous state is *ready*.

```
void wait() const;
```

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

future::wait_for

Blocks the current thread until the associated asynchronous state is *ready* or until a specified time interval has elapsed.

```
template <class Rep, class Period>  
future_status wait_for(const chrono::duration<Rep, Period>& Rel_time) const;
```

Parameters

Rel_time

A [chrono::duration](#) object that specifies a maximum time interval that the thread blocks.

Return Value

A [future_status](#) that indicates the reason for returning.

Remarks

An associated asynchronous state is ready only if its asynchronous provider has stored a return value or stored an exception.

future::wait_until

Blocks the current thread until the associated asynchronous state is *ready* or until after a specified time point.

```
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& Abs_time) const;
```

Parameters

Abs_time

A [chrono::time_point](#) object that specifies a time after which the thread can unblock.

Return Value

A [future_status](#) that indicates the reason for returning.

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

See also

[Header Files Reference](#)

[<future>](#)

future_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an exception object that can be thrown by methods of types that manage [future](#) objects.

Syntax

```
class future_error : public logic_error {  
public:  
    future_error(error_code code);  
  
    const error_code& code() const throw();  
  
    const char *what() const throw();  
  
};
```

Requirements

Header: <future>

Namespace: std

See also

[Header Files Reference](#)

[logic_error Class](#)

[error_code Class](#)

is_error_code_enum Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Specialization that indicates that [future_errc](#) is suitable for storing an [error_code](#).

Syntax

```
template <>
struct is_error_code_enum<Future_errc> : public true_type;
```

Requirements

Header: <future>

Namespace: std

See also

[Header Files Reference](#)

[<future>](#)

packaged_task Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Describes an *asynchronous provider* that is a call wrapper whose call signature is `Ty(ArgTypes...)`. Its *associated asynchronous state* holds a copy of its callable object in addition to the potential result.

Syntax

```
template <class>
class packaged_task;
```

Members

Public Constructors

NAME	DESCRIPTION
<code>packaged_task</code>	Constructs a <code>packaged_task</code> object.
<code>packaged_task::~~packaged_task</code> Destructor	Destroys a <code>packaged_task</code> object.

Public Methods

NAME	DESCRIPTION
<code>get_future</code>	Returns a <code>future</code> object that has the same associated asynchronous state.
<code>make_ready_at_thread_exit</code>	Calls the callable object that's stored in the associated asynchronous state and atomically stores the returned value.
<code>reset</code>	Replaces the associated asynchronous state.
<code>swap</code>	Exchanges the associated asynchronous state with that of a specified object.
<code>valid</code>	Specifies whether the object has an associated asynchronous state.

Public Operators

NAME	DESCRIPTION
<code>packaged_task::operator=</code>	Transfers an associated asynchronous state from a specified object.
<code>packaged_task::operator()</code>	Calls the callable object that's stored in the associated asynchronous state, atomically stores the returned value, and sets the state to <i>ready</i> .

NAME	DESCRIPTION
<code>packaged_task::operator bool</code>	Specifies whether the object has an associated asynchronous state.

Requirements

Header: `<future>`

Namespace: `std`

`packaged_task::get_future`

Returns an object of type `future<Ty>` that has the same *associated asynchronous state*.

```
future<Ty> get_future();
```

Remarks

If the `packaged_task` object does not have an associated asynchronous state, this method throws a `future_error` that has an error code of `no_state`.

If this method has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `future_already_retrieved`.

`packaged_task::make_ready_at_thread_exit`

Calls the callable object that's stored in the *associated asynchronous state* and atomically stores the returned value.

```
void make_ready_at_thread_exit(AngTypes... args);
```

Remarks

If the `packaged_task` object doesn't have an associated asynchronous state, this method throws a `future_error` that has an error code of `no_state`.

If this method or `make_ready_at_thread_exit` has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `promise_already_satisfied`.

Otherwise, this operator calls `INVOKE(fn, args..., Ty)`, where *fn* is the callable object that's stored in the associated asynchronous state. Any returned value is stored atomically as the returned result of the associated asynchronous state.

In contrast to `packaged_task::operator()`, the associated asynchronous state is not set to `ready` until after all thread-local objects in the calling thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state are not unblocked until the calling thread exits.

`packaged_task::operator=`

Transfers the *associated asynchronous state* from a specified object.

```
packaged_task& operator=(packaged_task&& Right);
```

Parameters

Right

A `packaged_task` object.

Return Value

`*this`

Remarks

After the operation, *Right* no longer has an associated asynchronous state.

packaged_task::operator()

Calls the callable object that's stored in the *associated asynchronous state*, atomically stores the returned value, and sets the state to *ready*.

```
void operator()(ArgTypes... args);
```

Remarks

If the `packaged_task` object doesn't have an associated asynchronous state, this method throws a `future_error` that has an error code of `no_state`.

If this method or `make_ready_at_thread_exit` has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `promise_already_satisfied`.

Otherwise, this operator calls `INVOKE(fn, args..., Ty)`, where *fn* is the callable object that's stored in the associated asynchronous state. Any returned value is stored atomically as the returned result of the associated asynchronous state, and the state is set to *ready*. As a result, any threads that are blocked on the associated asynchronous state become unblocked.

packaged_task::operator bool

Specifies whether the object has an `associated asynchronous state`.

```
operator bool() const noexcept;
```

Return Value

true if the object has an associated asynchronous state; otherwise, **false**.

packaged_task::packaged_task Constructor

Constructs a `packaged_task` object.

```
packaged_task() noexcept;
packaged_task(packaged_task&& Right) noexcept;
template <class Fn>
    explicit packaged_task(Fn&& fn);

template <class Fn, class Alloc>
    explicit packaged_task(
        allocator_arg_t, const Alloc& alloc, Fn&& fn);
```

Parameters

Right

A `packaged_task` object.

alloc

A memory allocator. For more information, see [<allocators>](#).

fn

A function object.

Remarks

The first constructor constructs a `packaged_task` object that has no *associated asynchronous state*.

The second constructor constructs a `packaged_task` object and transfers the associated asynchronous state from *Right*. After the operation, *Right* no longer has an associated asynchronous state.

The third constructor constructs a `packaged_task` object that has a copy of *fn* stored in its associated asynchronous state.

The fourth constructor constructs a `packaged_task` object that has a copy of *fn* stored in its associated asynchronous state, and uses `alloc` for memory allocation.

`packaged_task::~~packaged_task` Destructor

Destroys a `packaged_task` object.

```
~packaged_task();
```

Remarks

If the *associated asynchronous state* is not *ready*, the destructor stores a [future_error](#) exception that has an error code of `broken_promise` as the result in the associated asynchronous state, and any threads that are blocked on the associated asynchronous state become unblocked.

`packaged_task::reset`

Uses a new *associated asynchronous state* to replace the existing associated asynchronous state.

```
void reset();
```

Remarks

In effect, this method executes `*this = packaged_task(move(fn))`, where *fn* is the function object that's stored in the associated asynchronous state for this object. Therefore, the state of the object is cleared, and [get_future](#), [operator\(\)](#), and [make_ready_at_thread_exit](#) can be called as if on a newly-constructed object.

`packaged_task::swap`

Exchanges the associated asynchronous state with that of a specified object.

```
void swap(packaged_task& Right) noexcept;
```

Parameters

Right

A `packaged_task` object.

packaged_task::valid

Specifies whether the object has an `associated asynchronous state`.

```
bool valid() const;
```

Return Value

true if the object has an associated asynchronous state; otherwise, **false**.

See also

[Header Files Reference](#)

[<future>](#)

promise Class

10/31/2018 • 5 minutes to read • [Edit Online](#)

Describes an *asynchronous provider*.

Syntax

```
template <class Ty>
class promise;
```

Members

Public Constructors

NAME	DESCRIPTION
promise	Constructs a <code>promise</code> object.

Public Methods

NAME	DESCRIPTION
get_future	Returns a future associated with this promise.
set_exception	Atomically sets the result of this promise to indicate an exception.
set_exception_at_thread_exit	Atomically sets the result of this promise to indicate an exception, and delivers the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).
set_value	Atomically sets the result of this promise to indicate a value.
set_value_at_thread_exit	Atomically sets the result of this promise to indicate a value, and delivers the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).
swap	Exchanges the <i>associated asynchronous state</i> of this promise with that of a specified promise object.

Public Operators

NAME	DESCRIPTION
promise::operator=	Assignment of the shared state of this promise object.

Inheritance Hierarchy

promise

Requirements

Header: <future>

Namespace: std

promise::get_future

Returns a [future](#) object that has the same *associated asynchronous state* as this promise.

```
future<Ty> get_future();
```

Remarks

If the promise object is empty, this method throws a [future_error](#) that has an [error_code](#) of `no_state`.

If this method has already been called for a promise object that has the same associated asynchronous state, the method throws a `future_error` that has an [error_code](#) of `future_already_retrieved`.

promise::operator=

Transfers the *associated asynchronous state* from a specified `promise` object.

```
promise& operator=(promise&& Other) noexcept;
```

Parameters

Other

A `promise` object.

Return Value

`*this`

Remarks

This operator transfers the associated asynchronous state from *Other*. After the transfer, *Other* is *empty*.

promise::promise Constructor

Constructs a `promise` object.

```
promise();  
template <class Alloc>  
promise(allocator_arg_t, const Alloc& Al);  
promise(promise&& Other) noexcept;
```

Parameters

Al

A memory allocator. See [<allocators>](#) for more information.

Other

A `promise` object.

Remarks

The first constructor constructs an *empty* `promise` object.

The second constructor constructs an empty `promise` object and uses *Al* for memory allocation.

The third constructor constructs a `promise` object and transfers the associated asynchronous state from *Other*, and leaves *Other* empty.

promise::set_exception

Atomically stores an exception as the result of this `promise` object and sets the *associated asynchronous state* to *ready*.

```
void set_exception(exception_ptr Exc);
```

Parameters

Exc

An `exception_ptr` that's stored by this method as the exception result.

Remarks

If the `promise` object has no associated asynchronous state, this method throws a `future_error` that has an error code of `no_state`.

If `set_exception`, `set_exception_at_thread_exit`, `set_value`, or `set_value_at_thread_exit` has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

As a result of this method, any threads that are blocked on the associated asynchronous state become unblocked.

promise::set_exception_at_thread_exit

Atomically sets the result of this `promise` to indicate an exception, delivering the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).

```
void set_exception_at_thread_exit(exception_ptr Exc);
```

Parameters

Exc

An `exception_ptr` that's stored by this method as the exception result.

Remarks

If the promise object has no *associated asynchronous state*, this method throws a `future_error` that has an error code of `no_state`.

If `set_exception`, `set_exception_at_thread_exit`, `set_value`, or `set_value_at_thread_exit` has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

In contrast to `set_exception`, this method does not set the associated asynchronous state to ready until after all thread-local objects in the current thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state are not unblocked until the current thread exits.

promise::set_value

Atomically stores a value as the result of this `promise` object and sets the *associated asynchronous state* to *ready*.

```
void promise::set_value(const Ty& Val);
void promise::set_value(Ty&& Val);
void promise<Ty&>::set_value(Ty& Val);
void promise<void>::set_value();
```

Parameters

Val

The value to be stored as the result.

Remarks

If the `promise` object has no associated asynchronous state, this method throws a `future_error` that has an error code of `no_state`.

If `set_exception`, `set_exception_at_thread_exit`, `set_value`, or `set_value_at_thread_exit` has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

As a result of this method, any threads that are blocked on the associated asynchronous state become unblocked.

The first method also throws any exception that is thrown when *Val* is copied into the associated asynchronous state. In this situation, the associated asynchronous state is not set to ready.

The second method also throws any exception that is thrown when *Val* is moved into the associated asynchronous state. In this situation, the associated asynchronous state is not set to ready.

For the partial specialization `promise<Ty&>`, the stored value is in effect a reference to *Val*.

For the specialization `promise<void>`, no stored value exists.

promise::set_value_at_thread_exit

Atomically stores a value as the result of this `promise` object.

```
void promise::set_value_at_thread_exit(const Ty& Val);
void promise::set_value_at_thread_exit(Ty&& Val);
void promise<Ty&>::set_value_at_thread_exit(Ty& Val);
void promise<void>::set_value_at_thread_exit();
```

Parameters

Val

The value to be stored as the result.

Remarks

If the `promise` object has no *associated asynchronous state*, this method throws a `future_error` that has an error code of `no_state`.

If `set_exception`, `set_exception_at_thread_exit`, `set_value`, or `set_value_at_thread_exit` has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

In contrast to `set_value`, the associated asynchronous state is not set to ready until after all thread-local objects in the current thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state are not unblocked until the current thread exits.

The first method also throws any exception that is thrown when *Val* is copied into the associated asynchronous state.

The second method also throws any exception that is thrown when *Val* is moved into the associated asynchronous state.

For the partial specialization `promise<Ty&>`, the stored value is effectively a reference to *Val*.

For the specialization `promise<void>`, no stored value exists.

promise::swap

Exchanges the *associated asynchronous state* of this promise object with that of a specified object.

```
void swap(promise& Other) noexcept;
```

Parameters

Other

A `promise` object.

See also

[Header Files Reference](#)

shared_future Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an *asynchronous return object*. In contrast with a [future](#) object, an *asynchronous provider* can be associated with any number of `shared_future` objects.

Syntax

```
template <class Ty>
class shared_future;
```

Remarks

Do not call any methods other than `valid`, `operator=`, and the destructor on a `shared_future` object that's *empty*.

`shared_future` objects are not synchronized. Calling methods on the same object from multiple threads introduces a data race that has unpredictable results.

Members

Public Constructors

NAME	DESCRIPTION
shared_future	Constructs a <code>shared_future</code> object.

Public Methods

NAME	DESCRIPTION
get	Retrieves the result that's stored in the <i>associated asynchronous state</i> .
valid	Specifies whether the object is not empty.
wait	Blocks the current thread until the associated asynchronous state is ready.
wait_for	Blocks until the associated asynchronous state is ready or until the specified time has elapsed.
wait_until	Blocks until the associated asynchronous state is ready or until a specified point in time.

Public Operators

NAME	DESCRIPTION
shared_future::operator=	Assigns a new associated asynchronous state.

Requirements

Header: <future>

Namespace: std

shared_future::get

Retrieves the result that's stored in the *associated asynchronous state*.

```
const Ty& get() const;

Ty& get() const;

void get() const;
```

Remarks

If the result is an exception, the method rethrows it. Otherwise, the result is returned.

Before it retrieves the result, this method blocks the current thread until the associated asynchronous state is ready.

For the partial specialization `shared_future<Ty>`, the stored value is effectively a reference to the object that was passed to the *asynchronous provider* as the return value.

Because no stored value exists for the specialization `shared_future<void>`, the method returns **void**.

shared_future::operator=

Transfers an *associated asynchronous state* from a specified object.

```
shared_future& operator=(shared_future&& Right) noexcept;
shared_future& operator=(const shared_future& Right);
```

Parameters

Right

A `shared_future` object.

Return Value

`*this`

Remarks

For the first operator, *Right* no longer has an associated asynchronous state after the operation.

For the second method, *Right* maintains its associated asynchronous state.

shared_future::shared_future Constructor

Constructs a `shared_future` object.

```
shared_future() noexcept;
shared_future(future<Ty>&& Right) noexcept;
shared_future(shared_future&& Right) noexcept;
shared_future(const shared_future& Right);
```

Parameters

Right

A [future](#) or `shared_future` object.

Remarks

The first constructor constructs a `shared_future` object that has no *associated asynchronous state*.

The second and third constructors construct a `shared_future` object and transfer the associated asynchronous state from *Right*. *Right* no longer has an associated asynchronous state.

The fourth constructor constructs a `shared_future` object that has the same associated asynchronous state as *Right*.

shared_future::valid

Specifies whether the object has an *associated asynchronous state*.

```
bool valid() noexcept;
```

Return Value

true if the object has an associated asynchronous state; otherwise, **false**.

shared_future::wait

Blocks the current thread until the *associated asynchronous state* is *ready*.

```
void wait() const;
```

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

shared_future::wait_for

Blocks the current thread until the associated asynchronous state is *ready* or until a specified time has elapsed.

```
template <class Rep, class Period>
future_status wait_for(
    const chrono::duration<Rep, Period>& Rel_time) const;
```

Parameters

Rel_time

A [chrono::duration](#) object that specifies a maximum time interval that the thread blocks.

Return Value

A [future_status](#) that indicates the reason for returning.

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

shared_future::wait_until

Blocks the current thread until the associated asynchronous state is *ready* or until after a specified time point.

```
template <class Clock, class Duration>
future_status wait_until(
    const chrono::time_point<Clock, Duration>& Abs_time) const;
```

Parameters

Abs_time

A [chrono::time_point](#) object that specifies a time after which the thread can unblock.

Return Value

A [future_status](#) that indicates the reason for returning.

Remarks

An associated asynchronous state is ready only if its asynchronous provider has stored a return value or stored an exception.

See also

[Header Files Reference](#)

[<future>](#)

uses_allocator Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Specializations that always hold true.

Syntax

```
template <class Ty, class Alloc>
struct uses_allocator<promise<Ty>, Alloc> : true_type;
template <class Ty, class Alloc>
struct uses_allocator<packaged_task<Ty>, Alloc> : true_type;
```

Requirements

Header: <future>

Namespace: std

See also

[Header Files Reference](#)

[<future>](#)

<hash_map>

11/9/2018 • 2 minutes to read • [Edit Online](#)

NOTE

This header is obsolete. The alternative is [<unordered_map>](#).

Defines the container template classes `hash_map` and `hash_multimap` and their supporting templates.

Syntax

```
#include <hash_map>
```

Operators

HASH_MAP VERSION	HASH_MULTIMAP VERSION	DESCRIPTION
operator!= (hash_map)	operator!= (hash_multimap)	Tests if the <code>hash_map</code> or <code>hash_multimap</code> object on the left side of the operator is not equal to the <code>hash_map</code> or <code>hash_multimap</code> object on the right side.
operator== (hash_map)	operator== (hash_multimap)	Tests if the <code>hash_map</code> or <code>hash_multimap</code> object on the left side of the operator is equal to the <code>hash_map</code> or <code>hash_multimap</code> object on the right side.

Specialized Template Functions

HASH_MAP VERSION	HASH_MULTIMAP VERSION	DESCRIPTION
swap (hash_map)	swap (hash_multimap)	Exchanges the elements of two <code>hash_maps</code> or <code>hash_multimaps</code> .

Classes

CLASS	DESCRIPTION
hash_compare Class	Describes an object that can be used by any of the hash associative containers — <code>hash_map</code> , <code>hash_multimap</code> , <code>hash_set</code> , or <code>hash_multiset</code> — as a default <code>Traits</code> parameter object to order and hash the elements they contain.
value_compare Class	Provides a function object that can compare the elements of a <code>hash_map</code> by comparing the values of their keys to determine their relative order in the <code>hash_map</code> .
hash_map Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.

CLASS	DESCRIPTION
hash_multimap Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

Requirements

Header: `<hash_map>`

Namespace: `stdext`

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<hash_map> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap	swap (hash_map)

swap (hash_map)

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Exchanges the elements of two hash_maps.

```
void swap(  
    hash_map <Key, Type, Traits, Allocator>& left,  
    hash_map <Key, Type, Traits, Allocator>& right);
```

Parameters

right

The hash_map whose elements are to be exchanged with those of the map *left*.

left

The hash_map whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class hash_map to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

swap

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Exchanges the elements of two hash_multimaps.

```
void swap(  
    hash_multimap <Key, Type, Traits, Allocator>& left,  
    hash_multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

right

The hash_multimap whose elements are to be exchanged with those of the map *left*.

left

The hash_multimap whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class hash_multimap to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler.

When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

See also

[<hash_map>](#)

<hash_map> operators

10/31/2018 • 5 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator!= (multimap)</code>
<code>operator==</code>	<code>operator== (multimap)</code>

operator!=

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Tests if the hash_map object on the left side of the operator is not equal to the hash_map object on the right side.

```
bool operator!=(const hash_map <Key, Type, Traits, Allocator>& left, const hash_map <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `hash_map`.

right

An object of type `hash_map`.

Return Value

true if the hash_maps are not equal; **false** if hash_maps are equal.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. Two hash_maps are equal if they have the same number of elements and their respective elements have the same values.

Otherwise, they are unequal.

Members of the `<hash_map>` and `<hash_set>` header files in the [stdext Namespace](#).

Example

```

// hash_map_op_ne.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1, hm2, hm3;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        hm1.insert ( Int_Pair ( i, i ) );
        hm2.insert ( Int_Pair ( i, i * i ) );
        hm3.insert ( Int_Pair ( i, i ) );
    }

    if ( hm1 != hm2 )
        cout << "The hash_maps hm1 and hm2 are not equal." << endl;
    else
        cout << "The hash_maps hm1 and hm2 are equal." << endl;

    if ( hm1 != hm3 )
        cout << "The hash_maps hm1 and hm3 are not equal." << endl;
    else
        cout << "The hash_maps hm1 and hm3 are equal." << endl;
}

```

The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.

operator==

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Tests if the hash_map object on the left side of the operator is equal to the hash_map object on the right side.

```

bool operator==(const hash_map <Key, Type, Traits, Allocator>& left, const hash_map <Key, Type, Traits,
Allocator>& right);

```

Parameters

left

An object of type `hash_map`.

right

An object of type `hash_map`.

Return Value

true if the hash_map on the left side of the operator is equal to the hash_map on the right side of the operator;
otherwise **false**.

Remarks

The comparison between `hash_map` objects is based on a pairwise comparison of their elements. Two `hash_maps` are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hash_map_op_eq.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1, hm2, hm3;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        hm1.insert ( Int_Pair ( i, i ) );
        hm2.insert ( Int_Pair ( i, i * i ) );
        hm3.insert ( Int_Pair ( i, i ) );
    }

    if ( hm1 == hm2 )
        cout << "The hash_maps hm1 and hm2 are equal." << endl;
    else
        cout << "The hash_maps hm1 and hm2 are not equal." << endl;

    if ( hm1 == hm3 )
        cout << "The hash_maps hm1 and hm3 are equal." << endl;
    else
        cout << "The hash_maps hm1 and hm3 are not equal." << endl;
}
```

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

operator!= (hash_multimap)

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Tests if the `hash_multimap` object on the left side of the operator is not equal to the `hash_multimap` object on the right side.

```
bool operator!=(const hash_multimap <Key, Type, Traits, Allocator>& left, const hash_multimap <Key, Type,
Traits, Allocator>& right);
```

Parameters

left

An object of type `hash_multimap`.

right

An object of type `hash_multimap`.

Return Value

true if the hash_multimaps are not equal; **false** if hash_multimaps are equal.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. Two hash_multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hash_multimap_op_ne.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1, hm2, hm3;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        hm1.insert ( Int_Pair ( i, i ) );
        hm2.insert ( Int_Pair ( i, i * i ) );
        hm3.insert ( Int_Pair ( i, i ) );
    }

    if ( hm1 != hm2 )
        cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;
    else
        cout << "The hash_multimaps hm1 and hm2 are equal." << endl;

    if ( hm1 != hm3 )
        cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
    else
        cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
}
```

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

operator==(hash_multimap)

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Tests if the hash_multimap object on the left side of the operator is equal to the hash_multimap object on the right side.

```
bool operator==(const hash_multimap <Key, Type, Traits, Allocator>& left, const hash_multimap <Key, Type,
Traits, Allocator>& right);
```


Parameters

left

An object of type `hash_multimap`.

right

An object of type `hash_multimap`.

Return Value

true if the `hash_multimap` on the left side of the operator is equal to the `hash_multimap` on the right side of the operator; otherwise **false**.

Remarks

The comparison between `hash_multimap` objects is based on a pairwise comparison of their elements. Two `hash_multimaps` are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hash_multimap_op_eq.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2, hm3;
    int i;
    typedef pair<int, int> Int_Pair;

    for (i = 0; i < 3; i++)
    {
        hm1.insert(Int_Pair(i, i));
        hm2.insert(Int_Pair(i, i*i));
        hm3.insert(Int_Pair(i, i));
    }

    if ( hm1 == hm2 )
        cout << "The hash_multimaps hm1 and hm2 are equal." << endl;
    else
        cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;

    if ( hm1 == hm3 )
        cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
    else
        cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
}
```

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

See also

[`<hash_map>`](#)

hash_compare Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The template class describes an object that can be used by any of the hash associative containers — `hash_map`, `hash_multimap`, `hash_set`, or `hash_multiset` — as a default **Traits** parameter object to order and hash the elements they contain.

Syntax

```
class hash_compare { Traits comp; public: const size_t bucket_size = 4; const size_t min_buckets = 8;
hash_compare(); hash_compare(Traits pred); size_t operator()(const Key& key) const; bool operator()( const Key&
key1, const Key& key2) const; };
```

Remarks

Each hash associative container stores a hash traits object of type `Traits` (a template parameter). You can derive a class from a specialization of `hash_compare` to selectively override certain functions and objects, or you can supply your own version of this class if you meet certain minimum requirements. Specifically, for an object `hash_comp` of type `hash_compare<Key, Traits>`, the following behavior is required by the above containers:

- For all values `key` of type `Key`, the call **hash_comp**(`key`) serves as a hash function, which yields a distribution of values of type `size_t`. The function supplied by `hash_compare` returns `key`.
- For any value `key1` of type `Key` that precedes `key2` in the sequence and has the same hash value (value returned by the hash function), **hash_comp**(`key2`, `key1`) is false. The function must impose a total ordering on values of type `Key`. The function supplied by `hash_compare` returns `comp(key2, key1)`, where `comp` is a stored object of type `Traits` that you can specify when you construct the object `hash_comp`. For the default `Traits` parameter type `less<Key>`, sort keys never decrease in value.
- The integer constant `bucket_size` specifies the mean number of elements per "bucket" (hash-table entry) that the container should try not to exceed. It must be greater than zero. The value supplied by `hash_compare` is 4.
- The integer constant `min_buckets` specifies the minimum number of buckets to maintain in the hash table. It must be a power of two and greater than zero. The value supplied by `hash_compare` is 8.

Example

See examples for [hash_map::hash_map](#), [hash_multimap::hash_multimap](#), [hash_set::hash_set](#), and [hash_multiset::hash_multiset](#), for examples of how to declare and use `hash_compare`.

Requirements

Header: `<hash_map>`

Namespace: `stdext`

See also

[Thread Safety in the C++ Standard Library](#)
[C++ Standard Library Reference](#)

hash_map Class

11/14/2018 • 60 minutes to read • [Edit Online](#)

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Stores and retrieves data quickly from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.

Syntax

```
template <class Key,  
          class Type,  
          class Traits=hash_compare<Key, less<Key>>,  
          class Allocator=allocator<pair <const Key, Type>>>  
class hash_map
```

Parameters

Key

The key data type to be stored in the hash_map.

Type

The element data type to be stored in the hash_map.

Traits

The type which includes two function objects, one of class compare able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and `hash_compare<Key, less<Key>>` is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the hash_map's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<pair <const Key, Type>>`.

Remarks

The hash_map is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of

data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a `hash_map`, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The `hash_map` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of uniquely occurring keywords with associated string values providing, say, definitions. If, instead, the words had more than one correct definition, so that keys were not unique, then a `hash_multimap` would be the container of choice. If, on the other hand, just the list of words were being stored, then a `hash_set` would be the correct container. If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure.

The `hash_map` orders the sequence it controls by calling a stored hash *Traits* object of class `value_compare`. This stored object may be accessed by calling the member function `key_comp`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key>>`. Specifically, for all values *Key* of type *Key*, the call `Traits (Key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x, y)$ is a function object that has two argument objects `x` and `y` and a return value of **true** or **false**. An ordering imposed on a `hash_map` is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both $f(x, y)$ and $f(y, x)$ are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the `hash_map` class is a bidirectional iterator, but the class member functions `insert` and `hash_map` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the class

member functions.

Constructors

CONSTRUCTOR	DESCRIPTION
hash_map	Constructs a <code>hash_map</code> that is empty or that is a copy of all or part of some other <code>hash_map</code> .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	A type that represents the <code>allocator</code> class for the <code>hash_map</code> object.
const_iterator	A type that provides a bidirectional iterator that can read a <code>const</code> element in the <code>hash_map</code> .
const_pointer	A type that provides a pointer to a const element in a <code>hash_map</code> .
const_reference	A type that provides a reference to a const element stored in a <code>hash_map</code> for reading and performing const operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any const element in the <code>hash_map</code> .
difference_type	A signed integer type that can be used to represent the number of elements of a <code>hash_map</code> in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a <code>hash_map</code> .
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>hash_map</code> .
key_type	A type describes the sort key object that constitutes each element of the <code>hash_map</code> .
mapped_type	A type that represents the data type stored in a <code>hash_map</code> .
pointer	A type that provides a pointer to an element in a <code>hash_map</code> .
reference	A type that provides a reference to an element stored in a <code>hash_map</code> .
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed <code>hash_map</code> .
size_type	An unsigned integer type that can represent the number of elements in a <code>hash_map</code> .

TYPE NAME	DESCRIPTION
<code>value_type</code>	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the <code>hash_map</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>at</code>	Finds an element in a <code>hash_map</code> with a specified key value.
<code>begin</code>	Returns an iterator addressing the first element in the <code>hash_map</code> .
<code>cbegin</code>	Returns a const iterator addressing the first element in the <code>hash_map</code> .
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a <code>hash_map</code> .
<code>clear</code>	Erases all the elements of a <code>hash_map</code> .
<code>count</code>	Returns the number of elements in a <code>hash_map</code> whose key matches a parameter-specified key.
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed <code>hash_map</code> .
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed <code>hash_map</code> .
<code>emplace</code>	Inserts an element constructed in place into a <code>hash_map</code> .
<code>emplace_hint</code>	Inserts an element constructed in place into a <code>hash_map</code> , with a placement hint.
<code>empty</code>	Tests if a <code>hash_map</code> is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a <code>hash_map</code> .
<code>equal_range</code>	Returns a pair of iterators, respectively, to the first element in a <code>hash_map</code> with a key that is greater than a specified key and to the first element in the <code>hash_map</code> with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a <code>hash_map</code> from specified positions
<code>find</code>	Returns an iterator addressing the location of an element in a <code>hash_map</code> that has a key equivalent to a specified key.

MEMBER FUNCTION	DESCRIPTION
get_allocator	Returns a copy of the <code>allocator</code> object used to construct the <code>hash_map</code> .
insert	Inserts an element or a range of elements into a <code>hash_map</code> .
key_comp	Returns an iterator to the first element in a <code>hash_map</code> with a key value that is equal to or greater than that of a specified key.
lower_bound	Returns an iterator to the first element in a <code>hash_map</code> with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the <code>hash_map</code> .
rbegin	Returns an iterator addressing the first element in a reversed <code>hash_map</code> .
rend	Returns an iterator that addresses the location succeeding the last element in a reversed <code>hash_map</code> .
size	Returns the number of elements in the <code>hash_map</code> .
swap	Exchanges the elements of two <code>hash_map</code> s.
upper_bound	Returns an iterator to the first element in a <code>hash_map</code> that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a <code>hash_map</code> .

Operators

OPERATOR	DESCRIPTION
operator[]	Inserts an element into a <code>hash_map</code> with a specified key value.
hash_map::operator=	Replaces the elements of a <code>hash_map</code> with a copy of another <code>hash_map</code> .

Requirements

Header: `<hash_map>`

Namespace: `stdext`

`hash_map::allocator_type`

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that represents the allocator class for the hash_map object.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::allocator_type allocator_type;
```

Example

See example for [get_allocator](#) for an example using `allocator_type`.

hash_map::at

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Finds an element in a hash_map with a specified key value.

```
Type& at(const Key& key);  
  
const Type& at(const Key& key) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>key</i>	The key value of the element that is to be found.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value is not found, then the function throws an object of class [out_of_range Class](#).

Example


```

// hash_map_at.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef pair <const int, int> cInt2Int;
    hash_map <int, int> hm1;

    // Insert data values
    hm1.insert ( cInt2Int ( 1, 10 ) );
    hm1.insert ( cInt2Int ( 2, 20 ) );
    hm1.insert ( cInt2Int ( 3, 30 ) );

    cout << "The values of the mapped elements are:";
    for ( int i = 1 ; i <= hm1.size() ; i++ )
        cout << " " << hm1.at(i);
    cout << "." << endl;
}

```

hash_map::begin

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator addressing the first element in the hash_map.

```

const_iterator begin() const;

iterator begin();

```

Return Value

A bidirectional iterator addressing the first element in the hash_map or the location succeeding an empty hash_map.

Example

```

// hash_map_begin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: iterator hm1_Iter;
    hash_map <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 0, 0 ) );
    hm1.insert ( Int_Pair ( 1, 1 ) );
    hm1.insert ( Int_Pair ( 2, 4 ) );

    hm1_cIter = hm1.begin ( );
    cout << "The first element of hm1 is "
         << hm1_cIter -> first << "." << endl;

    hm1_Iter = hm1.begin ( );
    hm1.erase ( hm1_Iter );

    // The following 2 lines would err because the iterator is const
    // hm1_cIter = hm1.begin ( );
    // hm1.erase ( hm1_cIter );

    hm1_cIter = hm1.begin( );
    cout << "The first element of hm1 is now "
         << hm1_cIter -> first << "." << endl;
}

```

```

The first element of hm1 is 0.
The first element of hm1 is now 1.

```

hash_map::cbegin

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a const iterator addressing the first element in the hash_map.

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the [hash_map](#) or the location succeeding an empty

`hash_map`.

Example

```

// hash_map_cbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 2, 4 ) );

    hm1_cIter = hm1.cbegin ( );
    cout << "The first element of hm1 is "
         << hm1_cIter -> first << "." << endl;
}

```

The first element of hm1 is 2.

hash_map::cend

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a const iterator that addresses the location succeeding the last element in a hash_map.

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a [hash_map](#). If the

`hash_map` is empty, then `hash_map::cend == hash_map::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_map`.

The value returned by `cend` should not be dereferenced.

Example

```
// hash_map_cend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_cIter = hm1.cend( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is "
         << hm1_cIter -> second << "." << endl;
}
```

The value of last element of hm1 is 30.

hash_map::clear

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Erases all the elements of a hash_map.

```
void clear();
```

Remarks

Example

The following example demonstrates the use of the hash_map::clear member function.

```
// hash_map_clear.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_map is initially "
         << i << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_map after clearing is "
         << i << "." << endl;
}
```

The size of the hash_map is initially 2.
The size of the hash_map after clearing is 0.

hash_map::const_iterator

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a bidirectional iterator that can read a **const** element in the hash_map.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by hash_map points to elements that are objects of [value_type](#), that is of type `pair< const Key, Type >`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_iterator` `cIter` pointing to an element in a hash_map, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See example for [begin](#) for an example using `const_iterator`.

hash_map::const_pointer

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a pointer to a **const** element in a hash_map.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a hash_map object.

hash_map::const_reference

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a reference to a **const** element stored in a hash_map for reading and performing **const** operations.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_reference  
const_reference;
```

Remarks

Example

```

// hash_map_const_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( hm1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( hm1.begin( ) -> first );

    cout << "The key of the first element in the hash_map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( hm1.begin( ) -> second );

    cout << "The data value of the first element in the hash_map is "
         << Ref2 << "." << endl;
}

```

The key of the first element in the hash_map is 1.
The data value of the first element in the hash_map is 10.

hash_map::const_reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a bidirectional iterator that can read any **const** element in the hash_map.

```

typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_reverse_iterator
const_reverse_iterator;

```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_map in reverse.

The `const_reverse_iterator` defined by hash_map points to elements that are objects of [value_type](#), that is of type `pair < const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator` `crIter` pointing to an element in a hash_map, use the `->` operator.

To access the value of the key for the element, use `crIter -> first`, which is equivalent to `(* crIter).first`. To access the value of the mapped datum for the element, use `crIter -> second`, which is equivalent to `(* crIter).second`.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`.

hash_map::count

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns the number of elements in a hash_map whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key value of the elements to be matched from the hash_map.

Return Value

1 if the hash_map contains an element whose sort key matches the parameter key; 0 if the hash_map doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

`[lower_bound(key), upper_bound(key))`

which is 0 or 1 in the case of hash_map, which is a unique associative container.

Example

The following example demonstrates the use of the hash_map::count member function.


```

// hash_map_count.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair (1, 1));
    hm1.insert(Int_Pair (2, 1));
    hm1.insert(Int_Pair (1, 4));
    hm1.insert(Int_Pair (2, 1));

    // Keys must be unique in hash_map, so duplicates are ignored
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
}

```

```

The number of elements in hm1 with a sort key of 1 is: 1.
The number of elements in hm1 with a sort key of 2 is: 1.
The number of elements in hm1 with a sort key of 3 is: 0.

```

hash_map::crbegin

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a const iterator addressing the first element in a reversed hash_map.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [hash_map](#) or addressing what had been the last element in the unreversed `hash_map`.

Remarks

`crbegin` is used with a reversed hash_map just as [begin](#) is used with a `hash_map`.

With the return value of `crbegin`, the `hash_map` object cannot be modified.

`crbegin` can be used to iterate through a `hash_map` backwards.

Example

```

// hash_map_crbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_crIter = hm1.crbegin( );
    cout << "The first element of the reversed hash_map hm1 is "
         << hm1_crIter -> first << "." << endl;
}

```

The first element of the reversed hash_map hm1 is 3.

hash_map::crend

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_map.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [hash_map](#) (the location that had preceded the first element in the unreversed `hash_map`).

Remarks

`crend` is used with a reversed `hash_map` just as [hash_map::end](#) is used with a `hash_map`.

With the return value of `crend`, the `hash_map` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `hash_map`.

The value returned by `crend` should not be dereferenced.

Example

```

// hash_map_crend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_crIter = hm1.crend( );
    hm1_crIter--;
    cout << "The last element of the reversed hash_map hm1 is "
         << hm1_crIter -> first << "." << endl;
}

```

The last element of the reversed hash_map hm1 is 3.

hash_map::difference_type

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A signed integer type that can be used to represent the number of elements of a hash_map in a range between elements pointed to by iterators.

```

typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::difference_type
difference_type;

```

Example

```

// hash_map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>
#include <algorithm>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 3, 20 ) );

    // The following won't insert, because map keys are unique
    hm1.insert ( Int_Pair ( 2, 30 ) );

    hash_map <int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
    hm1_bIter = hm1.begin( );
    hm1_eIter = hm1.end( );

    // Count the number of elements in a hash_map
    hash_map <int, int>::difference_type df_count = 0;
    hm1_Iter = hm1.begin( );
    while ( hm1_Iter != hm1_eIter)
    {
        df_count++;
        hm1_Iter++;
    }

    cout << "The number of elements in the hash_map hm1 is: "
         << df_count << "." << endl;

    cout << "The keys of the mapped elements are:";
    for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
          hm1_Iter++)
        cout << " " << hm1_Iter-> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
          hm1_Iter++)
        cout << " " << hm1_Iter-> second;
    cout << "." << endl;
}

```

```

The number of elements in the hash_map hm1 is: 3.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 20.

```

hash_map::emplace

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Inserts an element constructed in place into a hash_map.

```
template <class ValTy>
pair <iterator, bool>
emplace(
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value used to move construct an element to be inserted into the <code>hash_map</code> unless the <code>hash_map</code> already contains that element (or, more generally, an element whose key is equivalently ordered).

Return Value

The `emplace` member function returns a pair whose bool component returns true if an insertion was made and false if the `hash_map` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **bool** component of a pair `pr` returned by this member function, use `pr.second`, and to dereference it, use `*(pr.second)`.

Remarks

The `hash_map::value_type` of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

```
// hash_map_emplace.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(move(is1));
    cout << "After the emplace insertion, hm1 contains:" << endl
         << " " << hm1.begin()->first
         << " => " << hm1.begin()->second
         << endl;
}
```

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_map::emplace_hint

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Inserts an element constructed in place into a `hash_map`, with a placement hint.

```
template <class ValTy>
iterator emplace_hint(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value used to move construct an element to be inserted into the hash_map unless the <code>hash_map</code> already contains that element (or, more generally, an element whose key is equivalently ordered).
<i>_Where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

The [hash_multimap::emplace](#) member function returns an iterator that points to the position where the new element was inserted into the `hash_map`, or where the existing element with equivalent ordering is located.

Remarks

The [hash_map::value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

```
// hash_map_emplace_hint.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(hm1.begin(), move(is1));
    cout << "After the emplace, hm1 contains:" << endl
         << " " << hm1.begin()->first
         << " => " << hm1.begin()->second
         << endl;
}
```

After the emplace insertion, hm1 contains:
1 => a

hash_map::empty

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Tests if a hash_map is empty.

```
bool empty() const;
```

Return Value

true if the hash_map is empty; **false** if the hash_map is nonempty.

Remarks

Example

```
// hash_map_empty.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1, hm2;

    typedef pair <int, int> Int_Pair;
    hm1.insert ( Int_Pair ( 1, 1 ) );

    if ( hm1.empty( ) )
        cout << "The hash_map hm1 is empty." << endl;
    else
        cout << "The hash_map hm1 is not empty." << endl;

    if ( hm2.empty( ) )
        cout << "The hash_map hm2 is empty." << endl;
    else
        cout << "The hash_map hm2 is not empty." << endl;
}
```

```
The hash_map hm1 is not empty.
The hash_map hm2 is empty.
```

hash_map::end

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator that addresses the location succeeding the last element in a hash_map.

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a `hash_map`. If the `hash_map` is empty, then `hash_map::end == hash_map::begin`.

Remarks

`end` is used to test whether an iterator has reached the end of its `hash_map`.

The value returned by `end` should not be dereferenced.

Example

```
// hash_map_end.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: iterator hm1_Iter;
    hash_map <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_cIter = hm1.end( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is "
         << hm1_cIter -> second << "." << endl;

    hm1_Iter = hm1.end( );
    hm1_Iter--;
    hm1.erase ( hm1_Iter );

    // The following 2 lines would err because the iterator is const
    // hm1_cIter = hm1.end ( );
    // hm1_cIter--;
    // hm1.erase ( hm1_cIter );

    hm1_cIter = hm1.end( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is now "
         << hm1_cIter -> second << "." << endl;
}
```

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

hash_map::equal_range

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a pair of iterators respectively to the first element in a hash_map with a key that is greater than a specified key and to the first element in the hash_map with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;  
  
pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key value to be compared with the sort key of an element from the hash_map being searched.

Return Value

A pair of iterators such that the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Remarks

Example

```

// hash_map_equal_range.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef hash_map <int, int> IntMap;
    IntMap hm1;
    hash_map <int, int> :: const_iterator hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
    p1 = hm1.equal_range( 2 );

    cout << "The lower bound of the element with "
         << "a key of 2 in the hash_map hm1 is: "
         << p1.first -> second << "." << endl;

    cout << "The upper bound of the element with "
         << "a key of 2 in the hash_map hm1 is: "
         << p1.second -> second << "." << endl;

    // Compare the upper_bound called directly
    hm1_RcIter = hm1.upper_bound( 2 );

    cout << "A direct call of upper_bound( 2 ) gives "
         << hm1_RcIter -> second << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 2 )." << endl;

    p2 = hm1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
        cout << "The hash_map hm1 doesn't have an element "
             << "with a key less than 40." << endl;
    else
        cout << "The element of hash_map hm1 with a key >= 40 is: "
             << p1.first -> first << "." << endl;
}

```

The lower bound of the element with a key of 2 in the hash_map hm1 is: 20.
 The upper bound of the element with a key of 2 in the hash_map hm1 is: 30.
 A direct call of upper_bound(2) gives 30,
 matching the 2nd element of the pair returned by equal_range(2).
 The hash_map hm1 doesn't have an element with a key less than 40.

hash_map::erase

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Removes an element or a range of elements in a hash_map from specified positions or removes elements that

match a specified key.

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

_Where

Position of the element to be removed from the hash_map.

first

Position of the first element removed from the hash_map.

last

Position just beyond the last element removed from the hash_map.

key

The key value of the elements to be removed from the hash_map.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_map if no such element exists.

For the third member function, returns the number of elements that have been removed from the hash_map.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the hash_map::erase member function.

```
// hash_map_erase.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2, hm3;
    hash_map<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_map<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;

    for (i = 1; i < 5; i++)
    {
        hm1.insert(Int_Pair (i, i));
        hm2.insert(Int_Pair (i, i*i));
        hm3.insert(Int_Pair (i, i-1));
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hm1.begin();
    hm1.erase(Iter1);

    cout << "After the 2nd element is deleted, the hash_map hm1 is:";
    for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
        cout << " " << *pIter << " second:";
```

```

        cout << " " << pIter -> second;
        cout << "." << endl;

        // The 2nd member function removes elements
        // in the range [ first, last)
        Iter1 = ++hm2.begin();
        Iter2 = --hm2.end();
        hm2.erase(Iter1, Iter2);

        cout << "After the middle two elements are deleted, "
              << "the hash_map hm2 is:";
        for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
            cout << " " << pIter -> second;
        cout << "." << endl;

        // The 3rd member function removes elements with a given key
        n = hm3.erase(2);

        cout << "After the element with a key of 2 is deleted,\n"
              << "the hash_map hm3 is:";
        for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
            cout << " " << pIter -> second;
        cout << "." << endl;

        // The 3rd member function returns the number of elements removed
        cout << "The number of elements removed from hm3 is: "
              << n << "." << endl;

        // The dereferenced iterator can also be used to specify a key
        Iter1 = ++hm3.begin();
        hm3.erase(Iter1);

        cout << "After another element with a key equal to that"
              << endl;
        cout << "of the 2nd element is deleted, "
              << "the hash_map hm3 is:";
        for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
            cout << " " << pIter -> second;
        cout << "." << endl;
    }
}

```

After the 2nd element is deleted, the hash_map hm1 is: 1 3 4.
 After the middle two elements are deleted, the hash_map hm2 is: 1 16.
 After the element with a key of 2 is deleted,
 the hash_map hm3 is: 0 2 3.
 The number of elements removed from hm3 is: 1.
 After another element with a key equal to that
 of the 2nd element is deleted, the hash_map hm3 is: 0 3.

hash_map::find

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator addressing the location of an element in a hash_map that has a key equivalent to a specified key.

```

iterator find(const Key& key);

const_iterator find(const Key& key) const;

```

Parameters

key

The key value to be matched by the sort key of an element from the hash_map being searched.

Return Value

An iterator that addresses the location of an element with a specified key, or the location succeeding the last element in the hash_map if no match is found for the key.

Remarks

`find` returns an iterator that addresses an element in the hash_map whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a [const_iterator](#), the hash_map object cannot be modified. If the return value of `find` is assigned to an [iterator](#), the hash_map object can be modified

Example

```
// hash_map_find.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;
    hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_RcIter = hm1.find( 2 );
    cout << "The element of hash_map hm1 with a key of 2 is: "
         << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    hm1_RcIter = hm1.find( 4 );

    if ( hm1_RcIter == hm1.end( ) )
        cout << "The hash_map hm1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of hash_map hm1 with a key of 4 is: "
             << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_map can be found
    // using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.find( hm1_AcIter -> first );
    cout << "The element of hm1 with a key matching "
         << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;
}
```

The element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.

hash_map::get_allocator

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a copy of the allocator object used to construct the hash_map.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_map.

Remarks

Allocators for the hash_map class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// hash_map_get_allocator.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int>::allocator_type hm1_Alloc;
    hash_map <int, int>::allocator_type hm2_Alloc;
    hash_map <int, double>::allocator_type hm3_Alloc;
    hash_map <int, int>::allocator_type hm4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    hash_map <int, int> hm1;
    hash_map <int, int> hm2;
    hash_map <int, double> hm3;

    hm1_Alloc = hm1.get_allocator( );
    hm2_Alloc = hm2.get_allocator( );
    hm3_Alloc = hm3.get_allocator( );

    cout << "The number of integers that can be allocated"
         << endl << "before free memory is exhausted: "
         << hm2.max_size( ) << "." << endl;

    cout << "The number of doubles that can be allocated"
         << endl << "before free memory is exhausted: "
         << hm3.max_size( ) << "." << endl;

    // The following line creates a hash_map hm4
    // with the allocator of hash_map hm1.
    hash_map <int, int> hm4( less<int>( ), hm1_Alloc );

    hm4_Alloc = hm4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated with the other
    if( hm1_Alloc == hm4_Alloc )
    {
        cout << "The allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
             << endl;
    }
}

```

hash_map::hash_map

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Constructs a hash_map that is empty or is a copy of all or part of some other hash_map.

```

hash_map();

explicit hash_map(
    const Traits& Comp);

hash_map(
    const Traits& Comp,
    const Allocator& Al);

hash_map(
    const hash_map& Right);

hash_map(
    hash_map&& Right);

hash_map(
    initializer_list<Type> IList);hash_map(initializer_list<Type> IList,
    const key_compare& Comp);

hash_map(
    initializer_list<Type> IList,
    const key_compare& Comp,
    const allocator_type& Al);

template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this hash_map object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the hash_map, which defaults to <code>hash_compare</code> .
<i>Right</i>	The hash_map of which the constructed map is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_map and can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_map.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the hash_map and that can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial hash_map, in addition, the second specifies the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifies the allocator type (*Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_map *Right*.

The next three constructors copy the range `[First, Last)` of a hash_map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

The last constructor moves the hash_map *Right*.

hash_map::insert

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Inserts an element or a range of elements into a hash_map.

```
pair <iterator, bool> insert(
    const value_type& val);

iterator insert(
    const_iterator _Where,
    const value_type& val);

template <class InputIterator>
void insert(
    InputIterator first,
    InputIterator last);

template <class ValTy>
pair <iterator, bool>
insert(
    ValTy&& val);

template <class ValTy>
iterator insert(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of an element to be inserted into the hash_map unless the hash_map already contains that element (or, more generally, an element whose key is equivalently ordered).
<i>_Where</i>	A hint regarding the place to start searching for the correct point of insertion.
<i>first</i>	The position of the first element to be copied from a hash_map.
<i>last</i>	The position just beyond the last element to be copied from a hash_map.

Return Value

The first `insert` member function returns a pair whose `bool` component returns true if an insertion was made and false if the hash_map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the `bool` component of a pair `pr` returned by this member function, use `pr.second`, and to dereference it, use `*(pr.second)`.

The second `insert` member function, the hint version, returns an iterator that points to the position where the new element was inserted into the hash_map.

The last two `insert` member functions behave the same as the first two, except that they move construct the inserted value.

Remarks

The `value_type` of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of `insert`, instead of logarithmic time, if the insertion point immediately follows `_Where`.

The third member function inserts the sequence of element values into a hash_map corresponding to each element addressed by an iterator of in the range `[First, Last)` of a specified set.

Example

```
// hash_map_insert.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int>::iterator hm1_pIter, hm2_pIter;

    hash_map<int, int> hm1, hm2;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(2, 20));
```

```

hm1.insert(Int_Pair(2, 20));
hm1.insert(Int_Pair(3, 30));
hm1.insert(Int_Pair(4, 40));

cout << "The original elements (Key => Value) of hm1 are:";
for (hm1_pIter = hm1.begin(); hm1_pIter != hm1.end(); hm1_pIter++)
    cout << endl << " " << hm1_pIter -> first << " => "
        << hm1_pIter->second;
cout << endl;

pair< hash_map<int,int>::iterator, bool > pr;
pr = hm1.insert(Int_Pair(1, 10));

if (pr.second == true)
{
    cout << "The element 10 was inserted in hm1 successfully."
        << endl;
}
else
{
    cout << "The element 10 already exists in hm1\n"
        << "with a key value of "
        << "((pr.first) -> first) = " << (pr.first)->first
        << "." << endl;
}

// The hint version of insert
hm1.insert(--hm1.end(), Int_Pair(5, 50));

cout << "After the insertions, the elements of hm1 are:";
for (hm1_pIter = hm1.begin(); hm1_pIter != hm1.end(); hm1_pIter++)
    cout << endl << hm1_pIter -> first << " => "
        << hm1_pIter->second;
cout << endl;

hm2.insert(Int_Pair(10, 100));

// The templated version inserting a range
hm2.insert( ++hm1.begin(), --hm1.end() );

cout << "After the insertions, the elements of hm2 are:";
for (hm2_pIter = hm2.begin(); hm2_pIter != hm2.end(); hm2_pIter++)
    cout << endl << hm2_pIter -> first << " => "
        << hm2_pIter->second;
cout << endl;

// The templated versions move constructing elements
hash_map<int, string> hm3, hm4;
pair<int, string> is1(1, "a"), is2(2, "b");

hm3.insert(move(is1));
cout << "After the move insertion, hm3 contains:" << endl
    << hm3.begin()->first
    << " => " << hm3.begin()->second
    << endl;

hm4.insert(hm4.begin(), move(is2));
cout << "After the move insertion, hm4 contains:" << endl
    << hm4.begin()->first
    << " => " << hm4.begin()->second
    << endl;
}

```

The original elements (Key => Value) of hm1 are:

```
1 => 10
2 => 20
3 => 30
4 => 40
```

The element 10 already exists in hm1
with a key value of ((pr.first) -> first) = 1.

After the insertions, the elements of hm1 are:

```
1 => 10
2 => 20
3 => 30
4 => 40
5 => 50
```

After the insertions, the elements of hm2 are:

```
2 => 20
10 => 100
3 => 30
4 => 40
```

After the move insertion, hm3 contains:

```
1 => a
```

After the move insertion, hm4 contains:

```
2 => b
```

hash_map::iterator

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a bidirectional iterator that can read or modify any element in a hash_map.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::iterator iterator;
```

Remarks

The `iterator` defined by hash_map points to elements that are objects of `value_type`, that is of type **pair<const Key, Type>**, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator** `Iter` pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `Iter -> first`, which is equivalent to `(* Iter).first`. To access the value of the mapped datum for the element, use `Iter -> second`, which is equivalent to `(* Iter).second`.

A type `iterator` can be used to modify the value of an element.

Example

See example for [begin](#) for an example of how to declare and use the `iterator`.

hash_map::key_comp

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Retrieves a copy of the comparison object used to order keys in a hash_map.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a hash_map uses to order its elements.

Remarks

The stored object defines the member function

bool operator(const Key& left, const Key& right);

that returns **true** if left precedes and is not equal to right in the sort order.

Example

```
// hash_map_key_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_map <int, int, hash_compare<int, less<int> > > > hm1;
    hash_map <int, int, hash_compare<int, less<int> > >::key_compare
        kc1 = hm1.key_comp( ) ;

    // Operator stored in kc1 tests order & returns bool value
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true,"
            << "\n where kc1 is the function object of hm1"
            << " of type key_compare." << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false"
            << "\n where kc1 is the function object of hm1"
            << " of type key_compare." << endl;
    }

    hash_map <int, int, hash_compare<int, greater<int> > > > hm2;
    hash_map <int, int, hash_compare<int, greater<int> > >
        ::key_compare kc2 = hm2.key_comp( );

    // Operator stored in kc2 tests order & returns bool value
    bool result2 = kc2( 2, 3 ) ;
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true,"
            << "\n where kc2 is the function object of hm2"
            << " of type key_compare." << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false,"
            << "\n where kc2 is the function object of hm2"
            << " of type key_compare." << endl;
    }
}
```

hash_map::key_compare

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the [hash_map Class](#) topic.

Example

See example for [key_comp](#) for an example of how to declare and use `key_compare`.

hash_map::key_type

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type describes the sort key object that constitutes each element of the hash_map.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `key`, see the Remarks section of the [hash_map Class](#) topic.

Example

See example for [value_type](#) for an example of how to declare and use `key_type`.

hash_map::lower_bound

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator to the first element in a hash_map with a key value that is equal to or greater than that of a specified key.

```
iterator lower_bound(const Key& key);  
  
const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key value to be compared with the sort key of an element from the hash_map being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of an element in a hash_map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_map if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the hash_map object cannot be modified. If the return value of `lower_bound` is assigned to a `iterator`, the hash_map object can be modified.

Remarks

Example

```
// hash_map_lower_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;
    hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_RcIter = hm1.lower_bound( 2 );
    cout << "The first element of hash_map hm1 with a key of 2 is: "
         << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    hm1_RcIter = hm1.lower_bound ( 4 );

    if ( hm1_RcIter == hm1.end( ) )
        cout << "The hash_map hm1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of hash_map hm1 with a key of 4 is: "
             << hm1_RcIter -> second << "." << endl;

    // An element at a specific location in the hash_map can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.lower_bound ( hm1_AcIter -> first );
    cout << "The element of hm1 with a key matching "
         << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;
}
```

The first element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.

hash_map::mapped_type

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that represents the data type stored in a hash_map.

```
typedef Type mapped_type;
```

Remarks

The type `mapped_type` is a synonym for the template parameter `Type`.

For more information on `Type` see the [hash_map Class](#) topic.

Example

See example for [value_type](#) for an example of how to declare and use `key_type`.

hash_map::max_size

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns the maximum length of the hash_map.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_map.

Remarks

Example

```
// hash_map_max_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int> :: size_type i;

    i = hm1.max_size( );
    cout << "The maximum possible length "
        << "of the hash_map is " << i << "."
        << endl << "(Magnitude is machine specific.);";
}
```

hash_map::operator[]

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Inserts an element into a `hash_map` with a specified key value.

```
Type& operator[](const Key& key);
```

```
Type& operator[](Key&& key);
```

Parameters

PARAMETER	DESCRIPTION
<i>key</i>	The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a `hash_map m` using

```
m[ key] = DataValue ;
```

where `DataValue` is the value of the `mapped_type` of the element with a key value of *key*.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a preexisting element or creating a new one. The member functions [find](#) and [insert](#) can be used to determine whether an element with a specified key is already present before an insertion.

Example

```

// hash_map_op_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef pair <const int, int> cInt2Int;
    hash_map <int, int> hm1;
    hash_map <int, int> :: iterator pIter;

    // Insert a data value of 10 with a key of 1
    // into a hash_map using the operator[] member function
    hm1[ 1 ] = 10;

    // Compare other ways to insert objects into a hash_map
    hm1.insert ( hash_map <int, int> :: value_type ( 2, 20 ) );
    hm1.insert ( cInt2Int ( 3, 30 ) );

    cout << "The keys of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;

    // If the key already exists, operator[]
    // changes the value of the datum in the element
    hm1[ 2 ] = 40;

    // operator[] will also insert the value of the data
    // type's default constructor if the value is unspecified
    hm1[5];

    cout << "The keys of the mapped elements are now:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are now:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;

    // operator[] will also insert by moving a key
    hash_map <string, int> hm2;
    string str("a");
    hm2[move(str)] = 1;
    cout << "The moved key is " << hm2.begin()->first
        << ", with value " << hm2.begin()->second << endl;
}

```

hash_map::operator=

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Replaces the elements of the `hash_map` with a copy of another `hash_map`.

```
hash_map& operator=(const hash_map& right);

hash_map& operator=(hash_map&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The hash_map Class being copied into the <code>hash_map</code> .

Remarks

After erasing any existing elements in a `hash_map`, `operator=` either copies or moves the contents of *right* into the `hash_map`.

Example

```
// hash_map_operator_as.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> v1, v2, v3;
    hash_map<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}
```

hash_map::pointer

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a pointer to an element in a `hash_map`.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a `hash_map` object.

hash_map::rbegin

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator addressing the first element in a reversed `hash_map`.

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed `hash_map` or addressing what had been the last element in the unreversed `hash_map`.

Remarks

`rbegin` is used with a reversed `hash_map` just as [begin](#) is used with a `hash_map`.

If the return value of `rbegin` is assigned to a [const_reverse_iterator](#), then the `hash_map` object cannot be modified. If the return value of `rbegin` is assigned to a [reverse_iterator](#), then the `hash_map` object can be modified.

`rbegin` can be used to iterate through a `hash_map` backwards.

Example

```

// hash_map_rbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: iterator hm1_Iter;
    hash_map <int, int> :: reverse_iterator hm1_rIter;
    hash_map <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_rIter = hm1.rbegin( );
    cout << "The first element of the reversed hash_map hm1 is "
         << hm1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a hash_map in a forward order
    cout << "The hash_map is: ";
    for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
        cout << hm1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a hash_map in a reverse order
    cout << "The reversed hash_map is: ";
    for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
        cout << hm1_rIter -> first << " ";
    cout << "." << endl;

    // A hash_map element can be erased by dereferencing to its key
    hm1_rIter = hm1.rbegin( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_map is "
         << hm1_rIter -> first << "." << endl;
}

```

```

The first element of the reversed hash_map hm1 is 3.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the first element in the reversed hash_map is 2.

```

hash_map::reference

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a reference to an element stored in a hash_map.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::reference reference;
```

Remarks

Example

```
// hash_map_reference.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( hm1.begin( ) -> first );

    // The following line would cause an error as the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( hm1.begin( ) -> first );

    cout << "The key of first element in the hash_map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( hm1.begin( ) -> second );

    cout << "The data value of first element in the hash_map is "
         << Ref2 << "." << endl;

    // The non-const_reference can be used to modify the
    // data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

```
The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The modified data value of first element is 15.
```

hash_map::rend

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator that addresses the location succeeding the last element in a reversed hash_map.

```
const_reverse_iterator rend() const;
```

```
reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_map (the location that had preceded the first element in the unreversed hash_map).

Remarks

`rend` is used with a reversed hash_map just as `end` is used with a hash_map.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the hash_map object cannot be modified.

If the return value of `rend` is assigned to a `reverse_iterator`, then the hash_map object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its hash_map.

The value returned by `rend` should not be dereferenced.

Example

```

// hash_map_rend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;

    hash_map <int, int> :: iterator hm1_Iter;
    hash_map <int, int> :: reverse_iterator hm1_rIter;
    hash_map <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "The last element of the reversed hash_map hm1 is "
         << hm1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a hash_map in a forward order
    cout << "The hash_map is: ";
    for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( );
          hm1_Iter++)
        cout << hm1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a hash_map in a reverse order
    cout << "The reversed hash_map is: ";
    for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( );
          hm1_rIter++)
        cout << hm1_rIter -> first << " ";
    cout << "." << endl;

    // A hash_map element can be erased by dereferencing to its key
    hm1_rIter = --hm1.rend( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed hash_map is "
         << hm1_rIter -> first << "." << endl;
}

```

```

The last element of the reversed hash_map hm1 is 1.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the last element in the reversed hash_map is 2.

```

hash_map::reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_map.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::reverse_iterator
reverse_iterator;
```

Remarks

A type `reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_map in reverse.

The `reverse_iterator` defined by hash_map points to elements that are objects of [value_type](#), that is of type **pair<const Key, Type>**, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `reverse_iterator` `rIter` pointing to an element in a hash_map, use the `->` operator.

To access the value of the key for the element, use `rIter -> first`, which is equivalent to `(* rIter).first`. To access the value of the mapped datum for the element, use `rIter -> second`, which is equivalent to `(* rIter).second`.

Example

See example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

hash_map::size

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns the number of elements in the hash_map.

```
size_type size() const;
```

Return Value

The current length of the hash_map.

Remarks

Example

The following example demonstrates the use of the hash_map::size member function.

```
// hash_map_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_map length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_map length is now " << i << "." << endl;
}
```

```
The hash_map length is 1.
The hash_map length is now 2.
```

hash_map::size_type

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

An unsigned integer type that can represent the number of elements in a hash_map.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See example for [size](#) for an example of how to declare and use `size_type`

hash_map::swap

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Exchanges the elements of two hash_maps.

```
void swap(hash_map& right);
```

Parameters

right

The argument hash_map providing the elements to be swapped with the target hash_map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_maps whose elements are being exchanged.

Example

```
// hash_map_swap.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1, hm2, hm3;
    hash_map <int, int>::iterator hm1_Iter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );
    hm2.insert ( Int_Pair ( 10, 100 ) );
    hm2.insert ( Int_Pair ( 20, 200 ) );
    hm3.insert ( Int_Pair ( 30, 300 ) );

    cout << "The original hash_map hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;

    // This is the member function version of swap
    // hm2 is said to be the argument hash_map;
    // hm1 is said to be the target hash_map
    hm1.swap( hm2 );

    cout << "After swapping with hm2, hash_map hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( hm1, hm3 );

    cout << "After swapping with hm3, hash_map hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;
}
```

The original hash_map hm1 is: 10 20 30.
After swapping with hm2, hash_map hm1 is: 100 200.
After swapping with hm3, hash_map hm1 is: 300.

hash_map::upper_bound

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns an iterator to the first element in a hash_map that with a key having a value that is greater than that of a

specified key.

```
iterator upper_bound(const Key& key);  
  
const_iterator upper_bound(const Key& key) const;
```

Parameters

key

The argument key value to be compared with the sort key value of an element from the hash_map being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of an element in a hash_map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the hash_map if no match is found for the key.

If the return value is assigned to a `const_iterator`, the hash_map object cannot be modified. If the return value is assigned to an `iterator`, the hash_map object can be modified.

Remarks

Example

```

// hash_map_upper_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map <int, int> hm1;
    hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_RcIter = hm1.upper_bound( 2 );
    cout << "The first element of hash_map hm1 with a key "
        << "greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end is returned
    hm1_RcIter = hm1.upper_bound ( 4 );

    if ( hm1_RcIter == hm1.end( ) )
        cout << "The hash_map hm1 doesn't have an element "
            << "with a key greater than 4." << endl;
    else
        cout << "The element of hash_map hm1 with a key > 4 is: "
            << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_map can be found
    // using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.begin( );
    hm1_RcIter = hm1.upper_bound ( hm1_AcIter -> first );
    cout << "The 1st element of hm1 with a key greater than that\n"
        << "of the initial element of hm1 is: "
        << hm1_RcIter -> second << "." << endl;
}

```

The first element of hash_map hm1 with a key greater than 2 is: 30.
 The hash_map hm1 doesn't have an element with a key greater than 4.
 The 1st element of hm1 with a key greater than that
 of the initial element of hm1 is: 20.

hash_map::value_comp

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

Returns a function object that determines the order of elements in a hash_map by comparing their key values.

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a hash_map uses to order its elements.

Remarks

For a hash_map m , if two elements $e1$ ($k1, d1$) and $e2$ ($k2, d2$) are objects of type `value_type`, where $k1$ and $k2$ are their keys of type `key_type` and $d1$ and $d2$ are their data of type `mapped_type`, then `m.value_comp()(e1, e2)` is equivalent to `m.key_comp()(k1, k2)`. A stored object defines the member function

```
bool operator(value_type& left, value_type& right);
```

which returns **true** if the key value of `left` precedes and is not equal to the key value of `right` in the sort order.

Example

```
// hash_map_value_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_map <int, int, hash_compare<int, less<int> > > > hm1;
    hash_map <int, int, hash_compare<int, less<int> > >
    ::value_compare vc1 = hm1.value_comp( );
    pair< hash_map<int,int>::iterator, bool > pr1, pr2;

    pr1= hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );
    pr2= hm1.insert ( hash_map <int, int> :: value_type ( 2, 5 ) );

    if( vc1( *pr1.first, *pr2.first ) == true )
    {
        cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
              << endl;
    }
    else
    {
        cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
              << endl;
    }

    if( vc1 ( *pr2.first, *pr1.first ) == true )
    {
        cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
              << endl;
    }
    else
    {
        cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
              << endl;
    }
}
```

hash_map::value_type

NOTE

This API is obsolete. The alternative is [unordered_map Class](#).

A type that represents the type of object stored in a hash_map.

```
typedef pair<const Key, Type> value_type;
```

Remarks

`value_type` is declared to be `pair<const key_type, mapped_type>` and not `pair<key_type, mapped_type>` because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example

```
// hash_map_value_type.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef pair <const int, int> cInt2Int;
    hash_map <int, int> hm1;
    hash_map <int, int> :: key_type key1;
    hash_map <int, int> :: mapped_type mapped1;
    hash_map <int, int> :: value_type value1;
    hash_map <int, int> :: iterator pIter;

    // value_type can be used to pass the correct type
    // explicitly to avoid implicit type conversion
    hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );

    // Compare other ways to insert objects into a hash_map
    hm1.insert ( cInt2Int ( 2, 20 ) );
    hm1[ 3 ] = 30;

    // Initializing key1 and mapped1
    key1 = ( hm1.begin( ) -> first );
    mapped1 = ( hm1.begin( ) -> second );

    cout << "The key of first element in the hash_map is "
          << key1 << "." << endl;

    cout << "The data value of first element in the hash_map is "
          << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout << "The keys of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

hash_multimap Class

11/14/2018 • 56 minutes to read • [Edit Online](#)

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

The container class `hash_multimap` is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

Syntax

```
template <class Key,
          class Type,
          class Traits=hash_compare <Key, less <Key>>,
          class Allocator=allocator <pair <const Key, Type>>>
class hash_multimap
```

Parameters

Key

The key data type to be stored in the `hash_multimap`.

Type

The element data type to be stored in the `hash_multimap`.

Traits

The type that includes two function objects, one of class *Traits* that is able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key>>` is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the `hash_multimap`'s allocation and deallocation of memory. This argument is optional, and the default value is

```
allocator<pair <const Key, Type>> .
```

Remarks

The `hash_multimap` is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Multiple, because its elements do not need to have a unique key, so that one key value may have many element data values associated with it.

- A pair associative container, because its element values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a `hash_multimap`, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The `hash_multimap` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words were not always uniquely defined. If, instead, the keywords were uniquely defined so that keys were unique, then a `hash_map` would be the container of choice. If, on the other hand, just the list of words were being stored, then a `hash_set` would be the correct container. If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure.

The `hash_multimap` orders the sequence it controls by calling a stored hash `Traits` object of type `value_compare`. This stored object may be accessed by calling the member function `key_comp`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key>>`. Specifically, for all values `key` of type `Key`, the call `Traits (Key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x, y)$ is a function object that has two argument objects `x` and `y` and a return value of **true** or **false**. An ordering imposed on a `hash_multimap` is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects `x` and `y` are defined to be equivalent when both $f(x, y)$ and $f(y, x)$ are **false**. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the `hash_multimap` class is a bidirectional iterator, but the class member functions `insert` and `hash_multimap` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own

hash_multimap of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal hash_multimap of functionality, but it is enough to be able to talk meaningfully about a range of iterators [First, Last) in the context of the member functions.

Constructors

CONSTRUCTOR	DESCRIPTION
hash_multimap	Constructs a list of a specific size or with elements of a specific value or with a specific <code>allocator</code> or as a copy of some other <code>hash_multimap</code> .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	A type that represents the <code>allocator</code> class for the <code>hash_multimap</code> object.
const_iterator	A type that provides a bidirectional iterator that can read a <code>const</code> element in the <code>hash_multimap</code> .
const_pointer	A type that provides a pointer to a const element in a <code>hash_multimap</code> .
const_reference	A type that provides a reference to a const element stored in a <code>hash_multimap</code> for reading and performing const operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any const element in the <code>hash_multimap</code> .
difference_type	A signed integer type that can be used to represent the number of elements of a <code>hash_multimap</code> in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a <code>hash_multimap</code> .
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>hash_multimap</code> .
key_type	A type that describes the sort key object that constitutes each element of the <code>hash_multimap</code> .
mapped_type	A type that represents the data type stored in a <code>hash_multimap</code> .
pointer	A type that provides a pointer to an element in a <code>hash_multimap</code> .

TYPE NAME	DESCRIPTION
reference	A type that provides a reference to an element stored in a <code>hash_multimap</code> .
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed <code>hash_multimap</code> .
size_type	An unsigned integer type that can represent the number of elements in a <code>hash_multimap</code> .
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the <code>hash_multimap</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
begin	Returns an iterator addressing the first element in the <code>hash_multimap</code> .
cbegin	Returns a const iterator addressing the first element in the <code>hash_multimap</code> .
cend	Returns a const iterator that addresses the location succeeding the last element in a <code>hash_multimap</code> .
clear	Erases all the elements of a <code>hash_multimap</code> .
count	Returns the number of elements in a <code>hash_multimap</code> whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed <code>hash_multimap</code> .
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed <code>hash_multimap</code> .
emplace	Inserts an element constructed in place into a <code>hash_multimap</code> .
emplace_hint	Inserts an element constructed in place into a <code>hash_multimap</code> , with a placement hint.
empty	Tests if a <code>hash_multimap</code> is empty.
end	Returns an iterator that addresses the location succeeding the last element in a <code>hash_multimap</code> .
equal_range	Returns an iterator that addresses the location succeeding the last element in a <code>hash_multimap</code> .

MEMBER FUNCTION	DESCRIPTION
erase	Removes an element or a range of elements in a <code>hash_multimap</code> from specified positions
find	Returns an iterator addressing the location of an element in a <code>hash_multimap</code> that has a key equivalent to a specified key.
get_allocator	Returns a copy of the <code>allocator</code> object used to construct the <code>hash_multimap</code> .
insert	Inserts an element or a range of elements into the <code>hash_multimap</code> at a specified position.
key_comp	Retrieves a copy of the comparison object used to order keys in a <code>hash_multimap</code> .
lower_bound	Returns an iterator to the first element in a <code>hash_multimap</code> that with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the <code>hash_multimap</code> .
rbegin	Returns an iterator addressing the first element in a reversed <code>hash_multimap</code> .
rend	Returns an iterator that addresses the location succeeding the last element in a reversed <code>hash_multimap</code> .
size	Specifies a new size for a <code>hash_multimap</code> .
swap	Exchanges the elements of two <code>hash_multimap</code> s.
upper_bound	Returns an iterator to the first element in a <code>hash_multimap</code> that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a <code>hash_multimap</code> .

Operators

OPERATOR	DESCRIPTION
hash_multimap::operator=	Replaces the elements of a <code>hash_multimap</code> with a copy of another <code>hash_multimap</code> .

Requirements

Header: `<hash_map>`

Namespace: `stdext`

`hash_multimap::allocator_type`

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that represents the allocator class for the hash_multimap object.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::allocator_type allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the [hash_multimap Class](#) topic.

Example

See the example for [get_allocator](#) for an example using `allocator_type`.

hash_multimap::begin

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator addressing the first element in the hash_multimap.

```
const_iterator begin() const;  
  
iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_multimap or the location succeeding an empty hash_multimap.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_multimap object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_multimap object can be modified.

Example

```

// hash_multimap_begin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: iterator hm1_Iter;
    hash_multimap <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 0, 0 ) );
    hm1.insert ( Int_Pair ( 1, 1 ) );
    hm1.insert ( Int_Pair ( 2, 4 ) );

    hm1_cIter = hm1.begin ( );
    cout << "The first element of hm1 is " << hm1_cIter -> first
         << "." << endl;

    hm1_Iter = hm1.begin ( );
    hm1.erase ( hm1_Iter );

    // The following 2 lines would err because the iterator is const
    // hm1_cIter = hm1.begin ( );
    // hm1.erase ( hm1_cIter );

    hm1_cIter = hm1.begin( );
    cout << "The first element of hm1 is now " << hm1_cIter -> first
         << "." << endl;
}

```

```

The first element of hm1 is 0.
The first element of hm1 is now 1.

```

hash_multimap::cbegin

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a const iterator addressing the first element in the hash_multimap.

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the [hash_multimap](#) or the location succeeding an empty `hash_multimap`.

Example

```
// hash_multimap_cbegin.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 2, 4 ) );

    hm1_cIter = hm1.cbegin ( );
    cout << "The first element of hm1 is "
         << hm1_cIter -> first << "." << endl;
}
```

The first element of hm1 is 2.

hash_multimap::cend

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a const iterator that addresses the location succeeding the last element in a hash_multimap.

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a [hash_multimap](#). If the `hash_multimap` is empty, then `hash_multimap::cend == hash_multimap::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its hash_multimap.

The value returned by `cend` should not be dereferenced.

Example


```
// hash_multimap_cend.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_cIter = hm1.cend( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is "
         << hm1_cIter -> second << "." << endl;
}
```

The value of last element of hm1 is 30.

hash_multimap::clear

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Erases all the elements of a hash_multimap.

```
void clear();
```

Remarks

Example

The following example demonstrates the use of the hash_multimap::clear member function.

```
// hash_multimap_clear.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_multimap is initially "
         << i << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_multimap after clearing is "
         << i << "." << endl;
}
```

The size of the hash_multimap is initially 2.
The size of the hash_multimap after clearing is 0.

hash_multimap::const_iterator

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a bidirectional iterator that can read a **const** element in the hash_multimap.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by hash_multimap points to objects of [value_type](#), which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair, and the value of the mapped element is available through the second member of the pair.

To dereference a `const_iterator` `cIter` pointing to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See the example for [begin](#) for an example using `const_iterator`.

hash_multimap::const_pointer

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a pointer to a **const** element in a hash_multimap.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a hash_multimap object.

hash_multimap::const_reference

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a reference to a **const** element stored in a hash_multimap for reading and performing **const** operations.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_reference  
const_reference;
```

Remarks

Example

```

// hash_multimap_const_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( hm1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( hm1.begin( ) -> first );

    cout << "The key of first element in the hash_multimap is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( hm1.begin() -> second );

    cout << "The data value of 1st element in the hash_multimap is "
         << Ref2 << "." << endl;
}

```

```

The key of first element in the hash_multimap is 1.
The data value of 1st element in the hash_multimap is 10.

```

hash_multimap::const_reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a bidirectional iterator that can read any **const** element in the hash_multimap.

```

typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_reverse_iterator
const_reverse_iterator;

```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_multimap in reverse.

The `const_reverse_iterator` defined by hash_multimap points to objects of [value_type](#), which are of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator` `crIter` pointing to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `crIter->first`, which is equivalent to `(*crIter).first`. To access the value of the mapped datum for the element, use `crIter->second`, which is equivalent to `(*crIter).second`.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`.

hash_multimap::count

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns the number of elements in a hash_multimap whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the hash_multimap.

Return Value

1 if the hash_multimap contains an element whose sort key matches the parameter key; 0 if the hash_multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

`[lower_bound (key), upper_bound (key))`

which have a key value *key*.

Example

The following example demonstrates the use of the hash_multimap::count member function.

```

// hash_multimap_count.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 1));
    hm1.insert(Int_Pair(1, 4));
    hm1.insert(Int_Pair(2, 1));

    // Elements do not need to have unique keys in hash_multimap,
    // so duplicates are allowed and counted
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
}

```

```

The number of elements in hm1 with a sort key of 1 is: 2.
The number of elements in hm1 with a sort key of 2 is: 2.
The number of elements in hm1 with a sort key of 3 is: 0.

```

hash_multimap::crbegin

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a const iterator addressing the first element in a reversed hash_multimap.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [hash_multimap](#) or addressing what had been the last element in the unreversed `hash_multimap`.

Remarks

`crbegin` is used with a reversed hash_multimap just as [hash_multimap::begin](#) is used with a `hash_multimap`.

With the return value of `crbegin`, the `hash_multimap` object cannot be modified.

`crbegin` can be used to iterate through a `hash_multimap` backwards.

Example

```
// hash_multimap_crbegin.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_crIter = hm1.crbegin( );
    cout << "The first element of the reversed hash_multimap hm1 is "
         << hm1_crIter -> first << "." << endl;
}
```

The first element of the reversed hash_multimap hm1 is 3.

hash_multimap::crend

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_multimap.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [hash_multimap](#) (the location that had preceded the first element in the unreversed `hash_multimap`).

Remarks

`crend` is used with a reversed hash_multimap just as [hash_multimap::end](#) is used with a hash_multimap.

With the return value of `crend`, the `hash_multimap` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its hash_multimap.

The value returned by `crend` should not be dereferenced.

Example

```
// hash_multimap_crend.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_crIter = hm1.crend( );
    hm1_crIter--;
    cout << "The last element of the reversed hash_multimap hm1 is "
         << hm1_crIter -> first << "." << endl;
}
```

The last element of the reversed hash_multimap hm1 is 3.

hash_multimap::difference_type

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A signed integer type that can be used to represent the number of elements of a hash_multimap in a range between elements pointed to by iterators.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::difference_type
difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range $[first, last)$ between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example


```

// hash_multimap_difference_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>
#include <algorithm>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(2, 20));
    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(3, 20));

    // The following will insert, because map keys
    // do not need to be unique
    hm1.insert(Int_Pair(2, 30));

    hash_multimap<int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
    hm1_bIter = hm1.begin();
    hm1_eIter = hm1.end();

    // Count the number of elements in a hash_multimap
    hash_multimap<int, int>::difference_type df_count = 0;
    hm1_Iter = hm1.begin();
    while (hm1_Iter != hm1_eIter)
    {
        df_count++;
        hm1_Iter++;
    }

    cout << "The number of elements in the hash_multimap hm1 is: "
         << df_count << "." << endl;

    cout << "The keys of the mapped elements are:";
    for (hm1_Iter= hm1.begin() ; hm1_Iter!= hm1.end();
         hm1_Iter++)
        cout << " " << hm1_Iter-> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for (hm1_Iter= hm1.begin() ; hm1_Iter!= hm1.end();
         hm1_Iter++)
        cout << " " << hm1_Iter-> second;
    cout << "." << endl;
}

```

The number of elements in the hash_multimap hm1 is: 4.
 The keys of the mapped elements are: 1 2 2 3.
 The values of the mapped elements are: 10 20 30 20.

hash_multimap::emplace

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Inserts an element constructed in place into a hash_multimap.

```
template <class ValTy>
iterator emplace(ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value used to move construct an element to be inserted into the hash_multimap .

Return Value

The `emplace` member function returns an iterator that points to the position where the new element was inserted.

Remarks

The [hash_multimap::value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

```
// hash_multimap_emplace.cpp
// compile with: /EHsc
#include<hash_multimap>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(move(is1));
    cout << "After the emplace, hm1 contains:" << endl
    << " " << hm1.begin()->first
    << " => " << hm1.begin()->second
    << endl;
}
```

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_multimap::emplace_hint

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Inserts an element constructed in place into a `hash_multimap`, with a placement hint.

```
template <class ValTy>
iterator emplace_hint(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value used to move construct an element to be inserted into the hash_multimap unless the <code>hash_multimap</code> already contains that element (or, more generally, an element whose key is equivalently ordered).
<i>_Where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

The [hash_multimap::emplace](#) member function returns an iterator that points to the position where the new element was inserted into the `hash_multimap`.

Remarks

The [hash_multimap::value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

```
// hash_multimap_emplace_hint.cpp
// compile with: /EHsc
#include<hash_multimap>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(hm1.begin(), move(is1));
    cout << "After the emplace insertion, hm1 contains:" << endl
         << " " << hm1.begin()->first
         << " => " << hm1.begin()->second
         << endl;
}
```

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_multimap::empty

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Tests if a `hash_multimap` is empty.

```
bool empty() const;
```

Return Value

true if the hash_multimap is empty; **false** if the hash_multimap is nonempty.

Remarks

Example

```
// hash_multimap_empty.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1, hm2;

    typedef pair <int, int> Int_Pair;
    hm1.insert ( Int_Pair ( 1, 1 ) );

    if ( hm1.empty( ) )
        cout << "The hash_multimap hm1 is empty." << endl;
    else
        cout << "The hash_multimap hm1 is not empty." << endl;

    if ( hm2.empty( ) )
        cout << "The hash_multimap hm2 is empty." << endl;
    else
        cout << "The hash_multimap hm2 is not empty." << endl;
}
```

```
The hash_multimap hm1 is not empty.
The hash_multimap hm2 is empty.
```

hash_multimap::end

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator that addresses the location succeeding the last element in a hash_multimap.

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multimap. If the hash_multimap is empty, then hash_multimap::end == hash_multimap::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_multimap.

The value returned by `end` should not be dereferenced.

Example

```
// hash_multimap_end.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: iterator hm1_Iter;
    hash_multimap <int, int> :: const_iterator hm1_cIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_cIter = hm1.end( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is "
         << hm1_cIter -> second << "." << endl;

    hm1_Iter = hm1.end( );
    hm1_Iter--;
    hm1.erase ( hm1_Iter );

    // The following 2 lines would err because the iterator is const
    // hm1_cIter = hm1.end( );
    // hm1_cIter--;
    // hm1.erase ( hm1_cIter );

    hm1_cIter = hm1.end( );
    hm1_cIter--;
    cout << "The value of last element of hm1 is now "
         << hm1_cIter -> second << "." << endl;
}
```

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

hash_multimap::equal_range

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a pair of iterators respectively to the first element in a hash_multimap with a key that is greater than a specified key and to the first element in the hash_multimap with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

A pair of iterators such that the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Remarks

Example

```
// hash_multimap_equal_range.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef hash_multimap <int, int> IntMMap;
    IntMMap hm1;
    hash_multimap <int, int> :: const_iterator hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
    p1 = hm1.equal_range( 2 );

    cout << "The lower bound of the element with a key of 2\n"
         << "in the hash_multimap hm1 is: "
         << p1.first -> second << "." << endl;

    cout << "The upper bound of the element with a key of 2\n"
         << "in the hash_multimap hm1 is: "
         << p1.second -> second << "." << endl;

    // Compare the upper_bound called directly
    hm1_RcIter = hm1.upper_bound( 2 );

    cout << "A direct call of upper_bound( 2 ) gives "
         << hm1_RcIter -> second << "," << endl
         << "matching the 2nd element of the pair "
         << "returned by equal_range( 2 )." << endl;

    p2 = hm1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
        cout << "The hash_multimap hm1 doesn't have an element "
             << "with a key less than 4." << endl;
    else
        cout << "The element of hash_multimap hm1 with a key >= 40 is: "
             << p1.first -> first << "." << endl;
}
```

The lower bound of the element with a key of 2
in the hash_multimap hm1 is: 20.
The upper bound of the element with a key of 2
in the hash_multimap hm1 is: 30.
A direct call of upper_bound(2) gives 30,
matching the 2nd element of the pair returned by equal_range(2).
The hash_multimap hm1 doesn't have an element with a key less than 4.

hash_multimap::erase

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Removes an element or a range of elements in a hash_multimap from specified positions or removes elements that match a specified key.

```
iterator erase(iterator _Where);  
  
iterator erase(iterator first, iterator last);  
  
size_type erase(const key_type& key);
```

Parameters

_Where

Position of the element to be removed from the hash_multimap.

first

Position of the first element removed from the hash_multimap.

last

Position just beyond the last element removed from the hash_multimap.

key

The key of the elements to be removed from the hash_multimap.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_multimap if no such element exists.

For the third member function, returns the number of elements that have been removed from the hash_multimap.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the hash_multimap::erase member function.

```
// hash_multimap_erase.cpp  
// compile with: /EHsc  
#include <hash_map>  
#include <iostream>  
  
int main()  
{
```

```

using namespace std;
using namespace stdext;
hash_multimap<int, int> hm1, hm2, hm3;
hash_multimap<int, int> :: iterator pIter, Iter1, Iter2;
int i;
hash_multimap<int, int>::size_type n;
typedef pair<int, int> Int_Pair;

for (i = 1; i < 5; i++)
{
    hm1.insert(Int_Pair (i, i) );
    hm2.insert(Int_Pair (i, i*i) );
    hm3.insert(Int_Pair (i, i-1) );
}

// The 1st member function removes an element at a given position
Iter1 = ++hm1.begin();
hm1.erase(Iter1);

cout << "After the 2nd element is deleted, "
    << "the hash_multimap hm1 is:";
for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
    cout << " " << pIter -> second;
cout << "." << endl;

// The 2nd member function removes elements
// in the range [ first, last)
Iter1 = ++hm2.begin();
Iter2 = --hm2.end();
hm2.erase(Iter1, Iter2);

cout << "After the middle two elements are deleted, "
    << "the hash_multimap hm2 is:";
for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
    cout << " " << pIter -> second;
cout << "." << endl;

// The 3rd member function removes elements with a given key
hm3.insert(Int_Pair (2, 5));
n = hm3.erase(2);

cout << "After the element with a key of 2 is deleted,\n"
    << "the hash_multimap hm3 is:";
for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
    cout << " " << pIter -> second;
cout << "." << endl;

// The 3rd member function returns the number of elements removed
cout << "The number of elements removed from hm3 is: "
    << n << "." << endl;

// The dereferenced iterator can also be used to specify a key
Iter1 = ++hm3.begin();
hm3.erase(Iter1);

cout << "After another element with a key equal to that of the"
    << endl;
cout << "2nd element is deleted, "
    << "the hash_multimap hm3 is:";
for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
    cout << " " << pIter -> second;
cout << "." << endl;
}

```


After the 2nd element is deleted, the hash_multimap hm1 is: 1 3 4.
After the middle two elements are deleted, the hash_multimap hm2 is: 1 16.
After the element with a key of 2 is deleted,
the hash_multimap hm3 is: 0 2 3.
The number of elements removed from hm3 is: 2.
After another element with a key equal to that of the
2nd element is deleted, the hash_multimap hm3 is: 0 3.

hash_multimap::find

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator addressing the first location of an element in a hash_multimap that has a key equivalent to a specified key.

```
iterator find(const Key& key);  
  
const_iterator find(const Key& key) const;
```

Parameters

key

The key to be matched by the sort key of an element from the hash_multimap being searched.

Return Value

An iterator that addresses the first location of an element with a specified key, or the location succeeding the last element in the hash_multimap if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multimap whose sort key is `equivalent` to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `find` is assigned to an `iterator`, the hash_multimap object can be modified.

Example

```

// hash_multimap_find.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(2, 20));
    hm1.insert(Int_Pair(3, 20));
    hm1.insert(Int_Pair(3, 30));

    hm1_RcIter = hm1.find(2);
    cout << "The element of hash_multimap hm1 with a key of 2 is: "
         << hm1_RcIter -> second << "." << endl;

    hm1_RcIter = hm1.find(3);
    cout << "The first element of hash_multimap hm1 with a key of 3 is: "
         << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end() is returned
    hm1_RcIter = hm1.find(4);

    if (hm1_RcIter == hm1.end())
        cout << "The hash_multimap hm1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of hash_multimap hm1 with a key of 4 is: "
             << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end();
    hm1_AcIter--;
    hm1_RcIter = hm1.find(hm1_AcIter -> first);
    cout << "The first element of hm1 with a key matching"
         << endl << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if (hm1_RcIter == --hm1.end())
        cout << "This is the last element of hash_multimap hm1."
             << endl;
    else
        cout << "This is not the last element of hash_multimap hm1."
             << endl;
}

```

```

The element of hash_multimap hm1 with a key of 2 is: 20.
The first element of hash_multimap hm1 with a key of 3 is: 20.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key matching
that of the last element is: 20.
This is not the last element of hash_multimap hm1.

```

hash_multimap::get_allocator

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns a copy of the allocator object used to construct the hash_multimap.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_multimap.

Remarks

Allocators for the hash_multimap class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// hash_multimap_get_allocator.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int>::allocator_type hm1_Alloc;
    hash_multimap <int, int>::allocator_type hm2_Alloc;
    hash_multimap <int, double>::allocator_type hm3_Alloc;
    hash_multimap <int, int>::allocator_type hm4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    hash_multimap <int, int> hm1;
    hash_multimap <int, int> hm2;
    hash_multimap <int, double> hm3;

    hm1_Alloc = hm1.get_allocator( );
    hm2_Alloc = hm2.get_allocator( );
    hm3_Alloc = hm3.get_allocator( );

    cout << "The number of integers that can be allocated"
         << endl << " before free memory is exhausted: "
         << hm2.max_size( ) << "." << endl;

    cout << "The number of doubles that can be allocated"
         << endl << " before free memory is exhausted: "
         << hm3.max_size( ) << "." << endl;

    // The following line creates a hash_multimap hm4
    // with the allocator of hash_multimap hm1.
    hash_multimap <int, int> hm4( less<int>( ), hm1_Alloc );

    hm4_Alloc = hm4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated by the other
    if( hm1_Alloc == hm4_Alloc )
    {
        cout << "The allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
             << endl;
    }
}

```

hash_multimap::hash_multimap

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Constructs a hash_multimap that is empty or is a copy of all or part of some other hash_multimap.

```

hash_multimap();

explicit hash_multimap(
    const Compare& Comp);

hash_multimap(
    const Compare& Comp,
    const Allocator& Al);

hash_multimap(
    const hash_multimap& Right);

hash_multimap(
    hash_multimap&& Right);

hash_multimap(
    initializer_list<Type> IList);

hash_multimap(
    initializer_list<Type> IList,
    const Compare& Comp);

hash_multimap(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last,
    const Compare& Comp);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last,
    const Compare& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this hash_multimap object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the map, which defaults to <code>Traits</code> .
<i>Right</i>	The map of which the constructed set is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list to copy from.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_multimap and that can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros are used to substitute alternative allocators.

All constructors initialize their hash_multimap.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the hash_multimap and can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial hash_multimap; the second specifies the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifies the allocator type (`_Al`) to be used. The keyword `explicit` suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_multimap `Right`.

The next three constructors copy the range `First, Last` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

The eighth constructor moves the hash_multimap `Right`.

The final three constructors use an initializer_list.

hash_multimap::insert

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Inserts an element or a range of elements into a hash_multimap.

```

iterator insert(
    const value_type& Val);

iterator insert(
    const_iterator Where,
    const value_type& Val); void insert(
    initializer_list<value_type> lList);

template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

template <class ValTy>
iterator insert(
    ValTy&& Val);

template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the hash_multimap unless it already contains that element, or more generally, unless it already contains an element whose key is equivalently ordered.
<i>Where</i>	A hint about where to start searching for the correct point of insertion.
<i>First</i>	The position of the first element to be copied from a map.
<i>Last</i>	The position just beyond the last element to be copied from a map.

Return Value

The first two `insert` member functions return an iterator that points to the position where the new element was inserted.

The third member function uses an `initializer_list` for the elements to be inserted.

The fourth member function inserts the sequence of element values into a map that corresponds to each element addressed by an iterator in the range `[First, Last)` of a specified set.

The last two `insert` member functions behave the same as the first two, except that they move-construct the inserted value.

Remarks

The `value_type` of an element is a pair, so that the value of an element will be an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of `insert`, instead of logarithmic time, if the insertion point immediately follows *Where*.

hash_multimap::iterator

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a bidirectional iterator that can read or modify any element in a hash_multimap.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::iterator iterator;
```

Remarks

The `iterator` defined by hash_multimap points to objects of `value_type`, which are of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator** `Iter` pointing to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `Iter -> first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter -> second`, which is equivalent to `(*Iter).first`.

A type `iterator` can be used to modify the value of an element.

Example

See the example for [begin](#) for an example of how to declare and use `iterator`.

hash_multimap::key_comp

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Retrieves a copy of the comparison object used to order keys in a hash_multimap.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a hash_multimap uses to order its elements.

Remarks

The stored object defines the member function

bool operator(const Key& `left` , const Key& `right`);

which returns **true** if `left` precedes and is not equal to `right` in the sort order.

Example


```

// hash_multimap_key_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_multimap <int, int, hash_compare<int, less<int>>> hm1;
    hash_multimap <int, int, hash_compare<int, less<int>>
        >::key_compare kc1 = hm1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true,\n"
            << "where kc1 is the function object of hm1.\n"
            << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false,\n"
            << "where kc1 is the function object of hm1.\n"
            << endl;
    }

    hash_multimap <int, int, hash_compare<int, greater<int>>> hm2;
    hash_multimap <int, int, hash_compare<int, greater<int>>
        >::key_compare kc2 = hm2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true,\n"
            << "where kc2 is the function object of hm2."
            << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false,\n"
            << "where kc2 is the function object of hm2."
            << endl;
    }
}

```

hash_multimap::key_compare

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_multimap.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [hash_multimap Class](#) topic.

Example

See the example for [key_comp](#) for an example of how to declare and use `key_compare`.

hash_multimap::key_type

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that describes the sort key object that constitutes each element of the hash_multimap.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

For more information on *Key*, see the Remarks section of the [hash_multimap Class](#) topic.

Example

See the example for [value_type](#) for an example of how to declare and use `key_compare`.

hash_multimap::lower_bound

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator to the first element in a hash_multimap with a key that is equal to or greater than a specified key.

```
iterator lower_bound(const Key& key);  
  
const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of an element in a hash_multimap with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `lower_bound` is assigned to an `iterator`, the hash_multimap object can be modified.

Remarks

Example

```

// hash_multimap_lower_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;
    hash_multimap <int, int> :: const_iterator hm1_AcIter,
        hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_RcIter = hm1.lower_bound( 2 );
    cout << "The element of hash_multimap hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

    hm1_RcIter = hm1.lower_bound( 3 );
    cout << "The first element of hash_multimap hm1 with a key of 3 is: "
        << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    hm1_RcIter = hm1.lower_bound( 4 );

    if ( hm1_RcIter == hm1.end( ) )
        cout << "The hash_multimap hm1 doesn't have an element "
            << "with a key of 4." << endl;
    else
        cout << "The element of hash_multimap hm1 with a key of 4 is: "
            << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end( );
    hm1_AcIter--;
    hm1_RcIter = hm1.lower_bound( hm1_AcIter -> first );
    cout << "The first element of hm1 with a key matching"
        << endl << "that of the last element is: "
        << hm1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( hm1_RcIter == --hm1.end( ) )
        cout << "This is the last element of hash_multimap hm1."
            << endl;
    else
        cout << "This is not the last element of hash_multimap hm1."
            << endl;
}

```

The element of hash_multimap hm1 with a key of 2 is: 20.
 The first element of hash_multimap hm1 with a key of 3 is: 20.
 The hash_multimap hm1 doesn't have an element with a key of 4.
 The first element of hm1 with a key matching
 that of the last element is: 20.
 This is not the last element of hash_multimap hm1.

hash_multimap::mapped_type

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that represents the data type stored in a hash_multimap.

```
typedef Type mapped_type;
```

Remarks

`mapped_type` is a synonym for the template parameter *Type*.

For more information on *Type* see the [hash_multimap Class](#) topic.

Example

See the example for [value_type](#) for an example of how to declare and use `key_type`.

hash_multimap::max_size

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns the maximum length of the hash_multimap.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_multimap.

Remarks

Example

```
// hash_multimap_max_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;
    hash_multimap <int, int> :: size_type i;

    i = hm1.max_size( );
    cout << "The maximum possible length "
         << "of the hash_multimap is " << i << "." << endl;
}
```

hash_multimap::operator=

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Replaces the elements of the `hash_multimap` with a copy of another `hash_multimap`.

```
hash_multimap& operator=(const hash_multimap& right);

hash_multimap& operator=(hash_multimap&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The hash_multimap being copied into the <code>hash_multimap</code> .

Remarks

After erasing any existing elements in a `hash_multimap`, `operator=` either copies or moves the contents of *right* into the `hash_multimap`.

Example

```
// hash_multimap_operator_as.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> v1, v2, v3;
    hash_multimap<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}
```

hash_multimap::pointer

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a pointer to an element in a hash_multimap.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a hash_multimap object.

hash_multimap::rbegin

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator addressing the first element in a reversed hash_multimap.

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multimap or addressing what had been the last element in the unreversed hash_multimap.

Remarks

`rbegin` is used with a reversed hash_multimap just as [begin](#) is used with a hash_multimap.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_multimap object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_multimap object can be modified.

`rbegin` can be used to iterate through a hash_multimap backwards.

Example

```

// hash_multimap_rbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: iterator hm1_Iter;
    hash_multimap <int, int> :: reverse_iterator hm1_rIter;
    hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_rIter = hm1.rbegin( );
    cout << "The first element of the reversed hash_multimap hm1 is "
         << hm1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a hash_multimap in a forward order
    cout << "The hash_multimap is: ";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++)
        cout << hm1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a hash_multimap in a reverse order
    cout << "The reversed hash_multimap is: ";
    for ( hm1_rIter = hm1.rbegin( ); hm1_rIter != hm1.rend( ); hm1_rIter++)
        cout << hm1_rIter -> first << " ";
    cout << "." << endl;

    // A hash_multimap element can be erased by dereferencing its key
    hm1_rIter = hm1.rbegin( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rbegin( );
    cout << "After the erasure, the first element\n"
         << "in the reversed hash_multimap is "
         << hm1_rIter -> first << "." << endl;
}

```

```

The first element of the reversed hash_multimap hm1 is 3.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the first element
in the reversed hash_multimap is 2.

```

hash_multimap::reference

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a reference to an element stored in a hash_multimap.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::reference reference;
```

Remarks

Example

```
// hash_multimap_reference.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( hm1.begin( ) -> first );

    // The following line would cause an error as the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( hm1.begin( ) -> first );

    cout << "The key of first element in the hash_multimap is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( hm1.begin( ) -> second );

    cout << "The data value of first element in the hash_multimap is "
         << Ref2 << "." << endl;

    //The non-const_reference can be used to modify the
    //data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

```
The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The modified data value of first element is 15.
```

hash_multimap::rend

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multimap.


```
const_reverse_iterator rend() const;  
  
reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multimap (the location that had preceded the first element in the unreversed hash_multimap).

Remarks

`rend` is used with a reversed hash_multimap just as `end` is used with a hash_multimap.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the hash_multimap object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the hash_multimap object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its hash_multimap.

The value returned by `rend` should not be dereferenced.

Example

```

// hash_multimap_rend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;

    hash_multimap <int, int> :: iterator hm1_Iter;
    hash_multimap <int, int> :: reverse_iterator hm1_rIter;
    hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "The last element of the reversed hash_multimap hm1 is "
         << hm1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a hash_multimap in a forward order
    cout << "The hash_multimap is: ";
    for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
        cout << hm1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a hash_multimap in a reverse order
    cout << "The reversed hash_multimap is: ";
    for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
        cout << hm1_rIter -> first << " ";
    cout << "." << endl;

    // A hash_multimap element can be erased by dereferencing its key
    hm1_rIter = --hm1.rend( );
    hm1.erase ( hm1_rIter -> first );

    hm1_rIter = hm1.rend( );
    hm1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed hash_multimap is "
         << hm1_rIter -> first << "." << endl;
}

```

```

The last element of the reversed hash_multimap hm1 is 1.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the last element in the reversed hash_multimap is 2.

```

hash_multimap::reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multimap.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::reverse_iterator
reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the `hash_multimap` in reverse.

The `reverse_iterator` defined by `hash_multimap` points to objects of `value_type`, which are of type `pair<const Key, Type>`. The value of the key is available through the first member `pair` and the value of the mapped element is available through the second member of the `pair`.

Example

See the example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

hash_multimap::size

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns the number of elements in the `hash_multimap`.

```
size_type size() const;
```

Return Value

The current length of the `hash_multimap`.

Remarks

Example

The following example demonstrates the use of the `hash_multimap::size` member function.

```
// hash_multimap_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_multimap length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_multimap length is now " << i << "." << endl;
}
```

```
The hash_multimap length is 1.  
The hash_multimap length is now 2.
```

hash_multimap::size_type

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

An unsigned integer type that counts the number of elements in a hash_multimap.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See the example for [size](#) for an example of how to declare and use `size_type`

hash_multimap::swap

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Exchanges the elements of two hash_multimaps.

```
void swap(hash_multimap& right);
```

Parameters

right

The hash_multimap providing the elements to be swapped or the hash_multimap whose elements are to be exchanged with those of the hash_multimap.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multimaps whose elements are being exchanged.

Example

```

// hash_multimap_swap.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1, hm2, hm3;
    hash_multimap <int, int>::iterator hm1_Iter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );
    hm2.insert ( Int_Pair ( 10, 100 ) );
    hm2.insert ( Int_Pair ( 20, 200 ) );
    hm3.insert ( Int_Pair ( 30, 300 ) );

    cout << "The original hash_multimap hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;

    // This is the member function version of swap
    hm1.swap( hm2 );

    cout << "After swapping with hm2, hash_multimap hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( hm1, hm3 );

    cout << "After swapping with hm3, hash_multimap hm1 is:";
    for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
        cout << " " << hm1_Iter -> second;
    cout << "." << endl;
}

```

The original hash_multimap hm1 is: 10 20 30.
 After swapping with hm2, hash_multimap hm1 is: 100 200.
 After swapping with hm3, hash_multimap hm1 is: 300.

hash_multimap::upper_bound

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

Returns an iterator to the first element in a hash_multimap with a key that is greater than a specified key.

```

iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;

```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of an element in a hash_multimap with a key that is greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of `upper_bound` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `upper_bound` is assigned to a `iterator`, the hash_multimap object can be modified.

Remarks

Example

```
// hash_multimap_upper_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap <int, int> hm1;
    hash_multimap <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair <int, int> Int_Pair;

    hm1.insert ( Int_Pair ( 1, 10 ) );
    hm1.insert ( Int_Pair ( 2, 20 ) );
    hm1.insert ( Int_Pair ( 3, 30 ) );
    hm1.insert ( Int_Pair ( 3, 40 ) );

    hm1_RcIter = hm1.upper_bound( 1 );
    cout << "The 1st element of hash_multimap hm1 with "
        << "a key greater than 1 is: "
        << hm1_RcIter -> second << "." << endl;

    hm1_RcIter = hm1.upper_bound( 2 );
    cout << "The first element of hash_multimap hm1\n"
        << "with a key greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    hm1_RcIter = hm1.lower_bound( 4 );

    if ( hm1_RcIter == hm1.end( ) )
        cout << "The hash_multimap hm1 doesn't have an element "
            << "with a key of 4." << endl;
    else
        cout << "The element of hash_multimap hm1 with a key of 4 is: "
            << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.begin( );
    hm1_RcIter = hm1.upper_bound( hm1_AcIter -> first );
    cout << "The first element of hm1 with a key greater than"
        << endl << "that of the initial element of hm1 is: "
        << hm1_RcIter -> second << "." << endl;
}
```

The 1st element of hash_multimap hm1 with a key greater than 1 is: 20.
The first element of hash_multimap hm1
with a key greater than 2 is: 30.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key greater than
that of the initial element of hm1 is: 20.

hash_multimap::value_comp

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

The member function returns a function object that determines the order of elements in a hash_multimap by comparing their key values.

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a hash_multimap uses to order its elements.

Remarks

For a hash_multimap m , if two elements $e1(k1, d1)$ and $e2(k2, d2)$ are objects of type [value_type](#), where $k1$ and $k2$ are their keys of type [key_type](#) and $d1$ and $d2$ are their data of type [mapped_type](#), then

`m.value_comp()(e1, e2)` is equivalent to `m.key_comp()(k1, k2)`. A stored object defines the member function

```
bool operator( value_type& left, value_type& right);
```

which returns **true** if the key value of `left` precedes and is not equal to the key value of `right` in the sort order.

Example

```

// hash_multimap_value_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_multimap <int, int, hash_compare<int, less<int>>> hm1;
    hash_multimap <int, int, hash_compare<int, less<int>>
        >::value_compare vc1 = hm1.value_comp( );
    hash_multimap <int,int>::iterator Iter1, Iter2;

    Iter1= hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );
    Iter2= hm1.insert ( hash_multimap <int, int> :: value_type ( 2, 5 ) );

    if( vc1( *Iter1, *Iter2 ) == true )
    {
        cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
            << endl;
    }
    else
    {
        cout << "The element ( 1,10 ) does "
            << "not precede the element ( 2,5 )."
            << endl;
    }

    if( vc1( *Iter2, *Iter1 ) == true )
    {
        cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
            << endl;
    }
    else
    {
        cout << "The element ( 2,5 ) does "
            << "not precede the element ( 1,10 )."
            << endl;
    }
}

```

hash_multimap::value_type

NOTE

This API is obsolete. The alternative is [unordered_multimap Class](#).

A type that represents the type of object stored in a hash_multimap.

```
typedef pair<const Key, Type> value_type;
```

Remarks

`value_type` is declared to be `pair<const key_type, mapped_type>` and not `pair<key_type, mapped_type>` because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example


```

// hash_multimap_value_type.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef pair <const int, int> cInt2Int;
    hash_multimap <int, int> hm1;
    hash_multimap <int, int> :: key_type key1;
    hash_multimap <int, int> :: mapped_type mapped1;
    hash_multimap <int, int> :: value_type value1;
    hash_multimap <int, int> :: iterator pIter;

    // value_type can be used to pass the correct type
    // explicitly to avoid implicit type conversion
    hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );

    // Compare another way to insert objects into a hash_multimap
    hm1.insert ( cInt2Int ( 2, 20 ) );

    // Initializing key1 and mapped1
    key1 = ( hm1.begin( ) -> first );
    mapped1 = ( hm1.begin( ) -> second );

    cout << "The key of first element in the hash_multimap is "
         << key1 << "." << endl;

    cout << "The data value of first element in the hash_multimap is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout << "The keys of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;
}

```

```

The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.

```

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

value_compare Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Provides a function object that can compare the elements of a `hash_map` by comparing the values of their keys to determine their relative order in the `hash_map`.

Syntax

```
class value_compare
: std::public binary_function<value_type, value_type, bool>
{
public:
    bool operator()(
        const value_type& left,
        const value_type& right) const
    {
        return (comp(left.first, right.first));
    }

protected:
    value_compare(const key_compare& c) : comp (c) { }
    key_compare comp;
};
```

Remarks

The comparison criteria provided by `value_compare` between `value_types` of whole elements contained by a `hash_map` is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object `comp` of type `key_compare` stored in the function object provided by `value_compare` to compare the sort-key components of two elements.

For `hash_sets` and `hash_multisets`, which are simple containers where the key values are identical to the element values, `value_compare` is equivalent to `key_compare`; for `hash_maps` and `hash_multimaps` they are not, because the value of the type `pair` elements is not identical to the value of the element's key.

Example

See the example for [hash_map::value_comp](#) for an example of how to declare and use `value_compare`.

Requirements

Header: `<hash_map>`

Namespace: `stdext`

See also

[binary_function Struct](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<hash_set>

11/9/2018 • 2 minutes to read • [Edit Online](#)

NOTE

This header is obsolete. The alternative is [<unordered_set>](#).

Defines the container template classes `hash_set` and `hash_multiset` and their supporting templates.

Syntax

```
#include <hash_set>
```

Remarks

Operators

HASH_SET VERSION	HASH_MULTISSET VERSION	DESCRIPTION
operator!= (hash_set)	operator!= (hash_multiset)	Tests if the <code>hash_set</code> or <code>hash_multiset</code> object on the left side of the operator is not equal to the <code>hash_set</code> or <code>hash_multiset</code> object on the right side.
operator== (hash_set)	operator== (hash_multiset)	Tests if the <code>hash_set</code> or <code>hash_multiset</code> object on the left side of the operator is equal to the <code>hash_set</code> or <code>hash_multiset</code> object on the right side.

Specialized Template Functions

HASH_SET VERSION	HASH_MULTISSET VERSION	DESCRIPTION
swap (hash_set)	swap (hash_multiset)	Exchanges the elements of two <code>hash_sets</code> or <code>hash_multisets</code> .

Classes

CLASS	DESCRIPTION
hash_compare Class	Describes an object that can be used by any of the hash associative containers — <code>hash_map</code> , <code>hash_multimap</code> , <code>hash_set</code> , or <code>hash_multiset</code> — as a default <code>Traits</code> parameter object to order and hash the elements they contain.
hash_set Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

CLASS	DESCRIPTION
hash_multiset Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

hash_set Class

11/14/2018 • 50 minutes to read • [Edit Online](#)

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

The container class `hash_set` is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

Syntax

```
template <class Key,  
          class Traits=hash_compare<Key, less<Key>>,  
          class Allocator=allocator<Key>>  
class hash_set
```

Parameters

Key

The element data type to be stored in the `hash_set`.

Traits

The type which includes two function objects, one of class `compare` that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the `hash_set`'s allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

The `hash_set` is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because `hash_set` is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The `hash_set` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a `hash_set` are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure. If values need to be attached to a list of unique key words, then a `hash_map` would be an appropriate structure to contain this data. If instead the keys are not unique, then a `hash_multimap` would be the container of choice.

The `hash_set` orders the sequence it controls by calling a stored hash `Traits` object of type `value_compare`. This stored object may be accessed by calling the member function `key_comp`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values `key` of type `Key`, the call `Trait(key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x, y)$ is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a `hash_set` is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both $f(x, y)$ and $f(y, x)$ are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the `hash_set` class is a bidirectional iterator, but the class member functions `insert` and `hash_set` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`first`, `last`) in the context of the class member functions.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>hash_set</code>	Constructs a <code>hash_set</code> that is empty or that is a copy of all or part of some other <code>hash_set</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for the <code>hash_set</code> object.
<code>const_iterator</code>	A type that provides a bidirectional iterator that can read a <code>const</code> element in the <code>hash_set</code> .
<code>const_pointer</code>	A type that provides a pointer to a const element in a <code>hash_set</code> .
<code>const_reference</code>	A type that provides a reference to a const element stored in a <code>hash_set</code> for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in the <code>hash_set</code> .
<code>difference_type</code>	A signed integer type that can be used to represent the number of elements of a <code>hash_set</code> in a range between elements pointed to by iterators.
<code>iterator</code>	A type that provides a bidirectional iterator that can read or modify any element in a <code>hash_set</code> .
<code>key_compare</code>	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>hash_set</code> .
<code>key_type</code>	A type that describes an object stored as an element of a <code>hash_set</code> in its capacity as sort key.
<code>pointer</code>	A type that provides a pointer to an element in a <code>hash_set</code> .
<code>reference</code>	A type that provides a reference to an element stored in a <code>hash_set</code> .
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed <code>hash_set</code> .
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a <code>hash_set</code> .
<code>value_compare</code>	A type that provides two function objects, a binary predicate of class <code>compare</code> that can compare two element values of a <code>hash_set</code> to determine their relative order and a unary predicate that hashes the elements.

TYPE NAME	DESCRIPTION
<code>value_type</code>	A type that describes an object stored as an element of a <code>hash_set</code> in its capacity as a value.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns an iterator that addresses the first element in the <code>hash_set</code> .
<code>cbegin</code>	Returns a const iterator addressing the first element in the <code>hash_set</code> .
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a <code>hash_set</code> .
<code>clear</code>	Erases all the elements of a <code>hash_set</code> .
<code>count</code>	Returns the number of elements in a <code>hash_set</code> whose key matches a parameter-specified key.
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed <code>hash_set</code> .
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed <code>hash_set</code> .
<code>emplace</code>	Inserts an element constructed in place into a <code>hash_set</code> .
<code>emplace_hint</code>	Inserts an element constructed in place into a <code>hash_set</code> , with a placement hint.
<code>empty</code>	Tests if a <code>hash_set</code> is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a <code>hash_set</code> .
<code>equal_range</code>	Returns a pair of iterators respectively to the first element in a <code>hash_set</code> with a key that is greater than a specified key and to the first element in the <code>hash_set</code> with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a <code>hash_set</code> from specified positions or removes elements that match a specified key.
<code>find</code>	Returns an iterator addressing the location of an element in a <code>hash_set</code> that has a key equivalent to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct the <code>hash_set</code> .

MEMBER FUNCTION	DESCRIPTION
insert	Inserts an element or a range of elements into a <code>hash_set</code> .
key_comp	Retrieves a copy of the comparison object used to order keys in a <code>hash_set</code> .
lower_bound	Returns an iterator to the first element in a <code>hash_set</code> with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the <code>hash_set</code> .
rbegin	Returns an iterator addressing the first element in a reversed <code>hash_set</code> .
rend	Returns an iterator that addresses the location succeeding the last element in a reversed <code>hash_set</code> .
size	Returns the number of elements in the <code>hash_set</code> .
swap	Exchanges the elements of two <code>hash_set</code> s.
upper_bound	Returns an iterator to the first element in a <code>hash_set</code> that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a <code>hash_set</code> .

Operators

OPERATOR	DESCRIPTION
hash_set::operator=	Replaces the elements of a <code>hash_set</code> with a copy of another <code>hash_set</code> .

Requirements

Header: `<hash_set>`

Namespace: `stdext`

hash_set::allocator_type

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that represents the allocator class for the `hash_set` object.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::allocator_type allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter *Allocator*.

For more information on *Allocator*, see the Remarks section of the [hash_set Class](#) topic.

Example

See example for [get_allocator](#) for an example that uses `allocator_type`.

hash_set::begin

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator that addresses the first element in the hash_set.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_set or the location succeeding an empty hash_set.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_set object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_set object can be modified.

Example

```
// hash_set_begin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::iterator hs1_Iter;
    hash_set <int>::const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_Iter = hs1.begin( );
    cout << "The first element of hs1 is " << *hs1_Iter << endl;

    hs1_Iter = hs1.begin( );
    hs1.erase( hs1_Iter );

    // The following 2 lines would err because the iterator is const
    // hs1_cIter = hs1.begin( );
    // hs1.erase( hs1_cIter );

    hs1_cIter = hs1.begin( );
    cout << "The first element of hs1 is now " << *hs1_cIter << endl;
}
```

```
The first element of hs1 is 1
The first element of hs1 is now 2
```

hash_set::cbegin

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a const iterator that addresses the first element in the hash_set.

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the [hash_set](#) or the location succeeding an empty `hash_set`.

Remarks

With the return value of `cbegin`, the elements in the `hash_set` object cannot be modified.

Example

```
// hash_set_cbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_cIter = hs1.cbegin( );
    cout << "The first element of hs1 is " << *hs1_cIter << endl;
}
```

```
The first element of hs1 is 1
```

hash_set::cend

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a const iterator that addresses the location succeeding the last element in a hash_set.

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a [hash_set](#). If the `hash_set` is empty, then `hash_set::cend == hash_set::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_set`. The value returned by `cend` should not be dereferenced.

Example

```
// hash_set_cend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int> :: const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_cIter = hs1.cend( );
    hs1_cIter--;
    cout << "The last element of hs1 is " << *hs1_cIter << endl;
}
```

```
The last element of hs1 is 3
```

hash_set::clear

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Erases all the elements of a `hash_set`.

```
void clear();
```

Remarks

Example

```
// hash_set_clear.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;

    hs1.insert( 1 );
    hs1.insert( 2 );

    cout << "The size of the hash_set is initially " << hs1.size( )
         << "." << endl;

    hs1.clear( );
    cout << "The size of the hash_set after clearing is "
         << hs1.size( ) << "." << endl;
}
```

The size of the hash_set is initially 2.
The size of the hash_set after clearing is 0.

hash_set::const_iterator

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a bidirectional iterator that can read a **const** element in the hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See example for [begin](#) for an example that uses `const_iterator`.

hash_set::const_pointer

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a pointer to a **const** element in a hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, a [const_iterator](#) should be used to access the elements in a **const** hash_set object.

hash_set::const_reference

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a reference to a **const** element stored in a hash_set for reading and performing **const** operations.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_reference const_reference;
```

Remarks

Example

```
// hash_set_const_ref.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;

    hs1.insert( 10 );
    hs1.insert( 20 );

    // Declare and initialize a const_reference &Ref1
    // to the 1st element
    const int &Ref1 = *hs1.begin( );

    cout << "The first element in the hash_set is "
         << Ref1 << "." << endl;

    // The following line would cause an error because the
    // const_reference cannot be used to modify the hash_set
    // Ref1 = Ref1 + 5;
}
```

```
The first element in the hash_set is 10.
```

hash_set::const_reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a bidirectional iterator that can read any **const** element in the hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_reverse_iterator
const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the `hash_set` in reverse.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`

hash_set::count

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns the number of elements in a `hash_set` whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the `hash_set`.

Return Value

1 if the `hash_set` contains an element whose sort key matches the parameter key.

0 if the `hash_set` does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[`lower_bound(key)`, `upper_bound(key)`).

Example

The following example demonstrates the use of the `hash_set::count` member function.

```
// hash_set_count.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1;
    hash_set<int>::size_type i;

    hs1.insert(1);
    hs1.insert(1);

    // Keys must be unique in hash_set, so duplicates are ignored.
    i = hs1.count(1);
    cout << "The number of elements in hs1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hs1.count(2);
    cout << "The number of elements in hs1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

```
The number of elements in hs1 with a sort key of 1 is: 1.
The number of elements in hs1 with a sort key of 2 is: 0.
```

hash_set::crbegin

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a const iterator addressing the first element in a reversed hash_set.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [hash_set](#) or addressing what had been the last element in the unreversed `hash_set`.

Remarks

`crbegin` is used with a reversed hash_set just as [hash_set::begin](#) is used with a hash_set.

With the return value of `crbegin`, the `hash_set` object cannot be modified.

`crbegin` can be used to iterate through a `hash_set` backwards.

Example


```
// hash_set_crbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::const_reverse_iterator hs1_crIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_crIter = hs1.crbegin( );
    cout << "The first element in the reversed hash_set is "
         << *hs1_crIter << "." << endl;
}
```

The first element in the reversed hash_set is 30.

hash_set::crend

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_set.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [hash_set](#) (the location that had preceded the first element in the unreversed `hash_set`).

Remarks

`crend` is used with a reversed `hash_set` just as [hash_set::end](#) is used with a `hash_set`.

With the return value of `crend`, the `hash_set` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `hash_set`.

Example

```
// hash_set_crend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::const_reverse_iterator hs1_crIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_crIter = hs1.crend( );
    hs1_crIter--;
    cout << "The last element in the reversed hash_set is "
         << *hs1_crIter << "." << endl;
}
```

The last element in the reversed hash_set is 10.

hash_set::difference_type

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A signed integer type that can be used to represent the number of elements of a hash_set in a range between elements pointed to by iterators.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container.

The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random access container, such as vector or deque.

Example

```

// hash_set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_set <int> hs1;
    hash_set <int>::iterator hs1_iter, hs1_bIter, hs1_eIter;

    hs1.insert( 20 );
    hs1.insert( 10 );
    hs1.insert( 20 ); // Won't insert as hash_set elements are unique

    hs1_bIter = hs1.begin( );
    hs1_eIter = hs1.end( );

    hash_set <int>::difference_type df_typ5, df_typ10, df_typ20;

    df_typ5 = count( hs1_bIter, hs1_eIter, 5 );
    df_typ10 = count( hs1_bIter, hs1_eIter, 10 );
    df_typ20 = count( hs1_bIter, hs1_eIter, 20 );

    // The keys, and hence the elements, of a hash_set are unique,
    // so there is at most one of a given value
    cout << "The number '5' occurs " << df_typ5
        << " times in hash_set hs1.\n";
    cout << "The number '10' occurs " << df_typ10
        << " times in hash_set hs1.\n";
    cout << "The number '20' occurs " << df_typ20
        << " times in hash_set hs1.\n";

    // Count the number of elements in a hash_set
    hash_set <int>::difference_type df_count = 0;
    hs1_iter = hs1.begin( );
    while ( hs1_iter != hs1_eIter)
    {
        df_count++;
        hs1_iter++;
    }

    cout << "The number of elements in the hash_set hs1 is: "
        << df_count << "." << endl;
}

```

```

The number '5' occurs 0 times in hash_set hs1.
The number '10' occurs 1 times in hash_set hs1.
The number '20' occurs 1 times in hash_set hs1.
The number of elements in the hash_set hs1 is: 2.

```

hash_set::emplace

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Inserts an element constructed in place into a hash_set.

```
template <class ValTy>
pair <iterator, bool>
emplace(
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of an element to be inserted into the hash_set unless the hash_set already contains that element or, more generally, an element whose key is equivalently ordered.

Return Value

The [emplace](#) member function returns a pair whose **bool** component returns **true** if an insertion was make and **false** if the [hash_set](#) already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

Remarks

Example

```
// hash_set_emplace.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<string> hs3;
    string str1("a");

    hs3.emplace(move(str1));
    cout << "After the emplace insertion, hs3 contains "
        << *hs3.begin() << "." << endl;
}
```

After the `emplace` insertion, `hs3` contains `a`.

hash_set::emplace_hint

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Inserts an element constructed in place into a `hash_set`.

```
template <class ValTy>
iterator emplace(
    const_iterator _where,
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of an element to be inserted into the hash_set unless the <code>hash_set</code> already contains that element or, more generally, an element whose key is equivalently ordered.
<i>_Where</i>	The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows <i>_Where</i> .)

Return Value

The [hash_set::emplace](#) member function returns an iterator that points to the position where the new element was inserted into the `hash_set`, or where the existing element with equivalent ordering is located.

Remarks

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

```
// hash_set_emplace_hint.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<string> hs3;
    string str1("a");

    hs3.insert(hs3.begin(), move(str1));
    cout << "After the emplace insertion, hs3 contains "
         << *hs3.begin() << "." << endl;
}
```

After the emplace insertion, hs3 contains a.

hash_set::empty

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Tests if a `hash_set` is empty.

```
bool empty() const;
```

Return Value

true if the `hash_set` is empty; **false** if the `hash_set` is nonempty.

Remarks

Example

```
// hash_set_empty.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1, hs2;
    hs1.insert ( 1 );

    if ( hs1.empty( ) )
        cout << "The hash_set hs1 is empty." << endl;
    else
        cout << "The hash_set hs1 is not empty." << endl;

    if ( hs2.empty( ) )
        cout << "The hash_set hs2 is empty." << endl;
    else
        cout << "The hash_set hs2 is not empty." << endl;
}
```

```
The hash_set hs1 is not empty.
The hash_set hs2 is empty.
```

hash_set::end

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator that addresses the location succeeding the last element in a hash_set.

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_set. If the hash_set is empty, then hash_set::end == hash_set::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_set. The value returned by `end` should not be dereferenced.

Example

```
// hash_set_end.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int> :: iterator hs1_Iter;
    hash_set <int> :: const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_Iter = hs1.end( );
    hs1_Iter--;
    cout << "The last element of hs1 is " << *hs1_Iter << endl;

    hs1.erase( hs1_Iter );

    // The following 3 lines would err because the iterator is const:
    // hs1_cIter = hs1.end( );
    // hs1_cIter--;
    // hs1.erase( hs1_cIter );

    hs1_cIter = hs1.end( );
    hs1_cIter--;
    cout << "The last element of hs1 is now " << *hs1_cIter << endl;
}
```

```
The last element of hs1 is 3
The last element of hs1 is now 2
```

hash_set::equal_range

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a pair of iterators respectively to the first element in a hash set with a key that is equal to a specified key and to the first element in the hash set with a key that is greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

A pair of iterators where the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair pr returned by the member function, use `pr.first`, and to dereference the

lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second`, and to dereference the upper bound iterator, use `*(pr.second)`.

Remarks

Example

```
// hash_set_equal_range.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef hash_set<int> IntHSet;
    IntHSet hs1;
    hash_set<int> :: const_iterator hs1_RcIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
    p1 = hs1.equal_range( 20 );

    cout << "The upper bound of the element with "
         << "a key of 20 in the hash_set hs1 is: "
         << *(p1.second) << "." << endl;

    cout << "The lower bound of the element with "
         << "a key of 20 in the hash_set hs1 is: "
         << *(p1.first) << "." << endl;

    // Compare the upper_bound called directly
    hs1_RcIter = hs1.upper_bound( 20 );
    cout << "A direct call of upper_bound( 20 ) gives "
         << *hs1_RcIter << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

    p2 = hs1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hs1.end( ) ) && ( p2.second == hs1.end( ) ) )
        cout << "The hash_set hs1 doesn't have an element "
             << "with a key greater than or equal to 40." << endl;
    else
        cout << "The element of hash_set hs1 with a key >= 40 is: "
             << *(p1.first) << "." << endl;
}
```

The upper bound of the element with a key of 20 in the hash_set hs1 is: 30.
The lower bound of the element with a key of 20 in the hash_set hs1 is: 20.
A direct call of upper_bound(20) gives 30,
matching the 2nd element of the pair returned by equal_range(20).
The hash_set hs1 doesn't have an element with a key greater than or equal to 40.

hash_set::erase

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Removes an element or a range of elements in a `hash_set` from specified positions or removes elements that match a specified key.

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

_Where

Position of the element to be removed from the `hash_set`.

first

Position of the first element removed from the `hash_set`.

last

Position just beyond the last element removed from the `hash_set`.

key

The key of the elements to be removed from the `hash_set`.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the `hash_set` if no such element exists. For the third member function, the number of elements that have been removed from the `hash_set`.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the `hash_set::erase` member function.

```
// hash_set_erase.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1, hs2, hs3;
    hash_set<int>::iterator pIter, Iter1, Iter2;
    int i;
    hash_set<int>::size_type n;

    for (i = 1; i < 5; i++)
    {
        hs1.insert (i);
        hs2.insert (i * i);
        hs3.insert (i - 1);
    }

    // The 1st member function removes an element at a given position
```

```

Iter1 = ++hs1.begin();
hs1.erase(Iter1);

cout << "After the 2nd element is deleted, the hash_set hs1 is:";
for (pIter = hs1.begin(); pIter != hs1.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 2nd member function removes elements
// in the range [ first, last)
Iter1 = ++hs2.begin();
Iter2 = --hs2.end();
hs2.erase(Iter1, Iter2);

cout << "After the middle two elements are deleted, "
    << "the hash_set hs2 is:";
for (pIter = hs2.begin(); pIter != hs2.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 3rd member function removes elements with a given key
n = hs3.erase(2);

cout << "After the element with a key of 2 is deleted, "
    << "the hash_set hs3 is:";
for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 3rd member function returns the number of elements removed
cout << "The number of elements removed from hs3 is: "
    << n << "." << endl;

// The dereferenced iterator can also be used to specify a key
Iter1 = ++hs3.begin();
hs3.erase(Iter1);

cout << "After another element (unique for hash_set) with a key "
    << endl;
cout << "equal to that of the 2nd element is deleted, "
    << "the hash_set hs3 is:";
for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;
}

```

After the 2nd element is deleted, the hash_set hs1 is: 1 3 4.
 After the middle two elements are deleted, the hash_set hs2 is: 16 4.
 After the element with a key of 2 is deleted, the hash_set hs3 is: 0 1 3.
 The number of elements removed from hs3 is: 1.
 After another element (unique for hash_set) with a key
 equal to that of the 2nd element is deleted, the hash_set hs3 is: 0 3.

hash_set::find

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key

The argument key to be matched by the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_set whose sort key is `equivalent` to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the hash_set object cannot be modified. If the return value of `find` is assigned to an `iterator`, the hash_set object can be modified.

Example

```
// hash_set_find.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_RcIter = hs1.find( 20 );
    cout << "The element of hash_set hs1 with a key of 20 is: "
         << *hs1_RcIter << "." << endl;

    hs1_RcIter = hs1.find( 40 );

    // If no match is found for the key, end( ) is returned
    if ( hs1_RcIter == hs1.end( ) )
        cout << "The hash_set hs1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of hash_set hs1 with a key of 40 is: "
             << *hs1_RcIter << "." << endl;

    // The element at a specific location in the hash_set can be found
    // by using a dereferenced iterator addressing the location
    hs1_AcIter = hs1.end( );
    hs1_AcIter--;
    hs1_RcIter = hs1.find( *hs1_AcIter );
    cout << "The element of hs1 with a key matching "
         << "that of the last element is: "
         << *hs1_RcIter << "." << endl;
}
```

```
The element of hash_set hs1 with a key of 20 is: 20.  
The hash_set hs1 doesn't have an element with a key of 40.  
The element of hs1 with a key matching that of the last element is: 30.
```

hash_set::get_allocator

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns a copy of the allocator object used to construct the hash_set.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_set to manage memory, which is the template parameter *Allocator*.

For more information on *Allocator*, see the Remarks section of the [hash_set Class](#) topic.

Remarks

Allocators for the hash_set class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// hash_set_get_allocator.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    // The following lines declare objects
    // that use the default allocator.
    hash_set <int, hash_compare <int, less<int> > > hs1;
    hash_set <int, hash_compare <int, greater<int> > > hs2;
    hash_set <double, hash_compare <double,
        less<double> >, allocator<double> > hs3;

    hash_set <int, hash_compare <int,
        greater<int> > >::allocator_type hs2_Alloc;
    hash_set <double>::allocator_type hs3_Alloc;
    hs2_Alloc = hs2.get_allocator( );

    cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs1.max_size( ) << "." << endl;

    cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs3.max_size( ) << "." << endl;

    // The following lines create a hash_set hs4
    // with the allocator of hash_set hs1.
    hash_set <int>::allocator_type hs4_Alloc;
    hash_set <int> hs4;
    hs4_Alloc = hs2.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated by the other
    if( hs2_Alloc == hs4_Alloc )
    {
        cout << "The allocators are interchangeable."
            << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
            << endl;
    }
}

```

hash_set::hash_set

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Constructs a `hash_set` that is empty or that is a copy of all or part of some other `hash_set`.

```

hash_set();

explicit hash_set(
    const Traits& Comp);

hash_set(
    const Traits& Comp,
    const Allocator& Al);

hash_set(
    const hash_set<Key, Traits, Allocator>& Right);

hash_set(
    hash_set&& Right);

hash_set(
    initializer_list<Type> IList);

hash_set(
    initializer_list<Type> IList,
    const Compare& Comp);

hash_set(
    initializer_list<value_type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this <code>hash_set</code> object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the <code>hash_set</code> , which defaults to <code>hash_compare</code> .
<i>Right</i>	The <code>hash_set</code> of which the constructed <code>hash_set</code> is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the `hash_set` and that can later be returned by calling `hash_set::get_allocator`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their `hash_sets`.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_set` and that can later be returned by calling `hash_set::key_comp`. For more information on `Traits` see the [hash_set Class](#) topic.

The first constructor creates an empty initial `hash_set`. The second specifies the type of comparison function (`Comp`) to be used in establishing the order of the elements, and the third explicitly specifies the allocator type (`Al`) to be used. The key word `explicit` suppresses certain kinds of automatic type conversion.

The fourth and fifth constructors specify a copy of the `hash_set` `Right` .

The last sixth, seventh, and eighth constructors use an `initializer_list` for the elements.

The last constructors copy the range [`First` , `Last`) of a `hash_set` with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

The eighth constructor moves the `hash_set` `Right` .

The actual order of elements in a `hash_set` container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the `set` container, where it was determined by the ordering function alone.

hash_set::insert

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Inserts an element or a range of elements into a `hash_set` .

```
pair<iterator, bool> insert(
    const value_type& Val);

iterator insert(
    iterator Where,
    const value_type& Val);

void insert(
    initializer_list<value_type> Ilist)
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);
```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the <code>hash_set</code> unless the <code>hash_set</code> already contains that element or, more generally, an element whose key is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows <code>_where</code> .)
<i>First</i>	The position of the first element to be copied from a <code>hash_set</code> .
<i>Last</i>	The position just beyond the last element to be copied from a <code>hash_set</code> .
<i>lList</i>	The initializer_list from which to copy the elements.

Return Value

The first `insert` member function returns a pair whose **bool** component returns **true** if an insertion was made and **false** if the `hash_set` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first` and to dereference it, use `*(pr.first)`. To access the **bool** component of a pair `pr` returned by this member function, use `pr.second`, and to dereference it, use `*(pr.second)`.

The second `insert` member function returns an iterator that points to the position where the new element was inserted into the `hash_set`.

Remarks

The third member function inserts the elements in an initializer_list.

The third member function inserts the sequence of element values into a `hash_set` corresponding to each element addressed by an iterator of in the range [`First`, `Last`) of a specified `hash_set`.

hash_set::iterator

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a bidirectional iterator that can read or modify any element in a `hash_set`.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::iterator iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for [begin](#) for an example of how to declare and use `iterator`.

hash_set::key_comp

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Retrieves a copy of the hash traits object used to hash and order element key values in a `hash_set`.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a `hash_set` uses to order its elements, which is the template parameter *Traits*.

For more information on *Traits* see the [hash_set Class](#) topic.

Remarks

The stored object defines the member function:

```
bool operator( const Key& _xVal, const Key& _yVal );
```

which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both [key_compare](#) and [value_compare](#) are synonyms for the template parameter *Traits*. Both types are provided for the `hash_set` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

```

// hash_set_key_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_set <int, hash_compare < int, less<int> > > hs1;
    hash_set<int, hash_compare < int, less<int> > >::key_compare kc1
        = hs1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
              << "where kc1 is the function object of hs1."
              << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
              << "where kc1 is the function object of hs1."
              << endl;
    }

    hash_set <int, hash_compare < int, greater<int> > > hs2;
    hash_set<int, hash_compare < int, greater<int> > >::key_compare
        kc2 = hs2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if(result2 == true)
    {
        cout << "kc2( 2,3 ) returns value of true, "
              << "where kc2 is the function object of hs2."
              << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
              << "where kc2 is the function object of hs2."
              << endl;
    }
}

```

hash_set::key_compare

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `hash_set`.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [hash_set Class](#) topic.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for `key_comp` for an example of how to declare and use `key_compare`.

hash_set::key_type

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that describes an object stored as an element of a hash_set in its capacity as sort key.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

For more information on *Key*, see the Remarks section of the [hash_set Class](#) topic.

Note that both `key_type` and `value_type` are synonyms for the template parameter *Key*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

See the example for `value_type` for an example of how to declare and use `key_type`.

hash_set::lower_bound

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(const Key& key) const;  
  
iterator lower_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

Example

```

// hash_set_lower_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_RcIter = hs1.lower_bound( 20 );
    cout << "The element of hash_set hs1 with a key of 20 is: "
         << *hs1_RcIter << "." << endl;

    hs1_RcIter = hs1.lower_bound( 40 );

    // If no match is found for the key, end( ) is returned
    if ( hs1_RcIter == hs1.end( ) )
        cout << "The hash_set hs1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of hash_set hs1 with a key of 40 is: "
             << *hs1_RcIter << "." << endl;

    // An element at a specific location in the hash_set can be found
    // by using a dereferenced iterator that addresses the location
    hs1_AcIter = hs1.end( );
    hs1_AcIter--;
    hs1_RcIter = hs1.lower_bound( *hs1_AcIter );
    cout << "The element of hs1 with a key matching "
         << "that of the last element is: "
         << *hs1_RcIter << "." << endl;
}

```

The element of hash_set hs1 with a key of 20 is: 20.
 The hash_set hs1 doesn't have an element with a key of 40.
 The element of hs1 with a key matching that of the last element is: 30.

hash_set::max_size

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns the maximum length of the hash_set.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_set.

Remarks

Example

```
// hash_set_max_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::size_type i;

    i = hs1.max_size( );
    cout << "The maximum possible length "
         << "of the hash_set is " << i << "." << endl;
}
```

hash_set::operator=

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Replaces the elements of the hash_set with a copy of another hash_set.

```
hash_set& operator=(const hash_set& right);

hash_set& operator=(hash_set&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The hash_set being copied into the <code>hash_set</code> .

Remarks

After erasing any existing elements in a `hash_set`, `operator=` either copies or moves the contents of *right* into the `hash_set`.

Example

```

// hash_set_operator_as.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<int> v1, v2, v3;
    hash_set<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter << " ";
    cout << endl;
}

```

hash_set::pointer

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a pointer to an element in a hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a hash_set object.

hash_set::rbegin

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator addressing the first element in a reversed hash_set.

```
const_reverse_iterator rbegin() const;
```

```
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_set or addressing what had been the last element in the unreversed hash_set.

Remarks

`rbegin` is used with a reversed hash_set just as `begin` is used with a hash_set.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_set object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_set object can be modified.

`rbegin` can be used to iterate through a hash_set backwards.

Example

```

// hash_set_rbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::iterator hs1_Iter;
    hash_set <int>::reverse_iterator hs1_rIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_rIter = hs1.rbegin( );
    cout << "The first element in the reversed hash_set is "
         << *hs1_rIter << "." << endl;

    // begin can be used to start an iteration
    // through a hash_set in a forward order
    cout << "The hash_set is: ";
    for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( );
          hs1_Iter++ )
        cout << *hs1_Iter << " ";
    cout << endl;

    // rbegin can be used to start an iteration
    // through a hash_set in a reverse order
    cout << "The reversed hash_set is: ";
    for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
          hs1_rIter++ )
        cout << *hs1_rIter << " ";
    cout << endl;

    // A hash_set element can be erased by dereferencing to its key
    hs1_rIter = hs1.rbegin( );
    hs1.erase ( *hs1_rIter );

    hs1_rIter = hs1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_set is "<< *hs1_rIter << "."
         << endl;
}

```

```

The first element in the reversed hash_set is 30.
The hash_set is: 10 20 30
The reversed hash_set is: 30 20 10
After the erasure, the first element in the reversed hash_set is 20.

```

hash_set::reference

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a reference to an element stored in a hash_set.


```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::reference reference;
```

Remarks

Example

```
// hash_set_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;

    hs1.insert( 10 );
    hs1.insert( 20 );

    // Declare and initialize a reference &Ref1 to the 1st element
    int &Ref1 = *hs1.begin( );

    cout << "The first element in the hash_set is "
         << Ref1 << "." << endl;

    // The value of the 1st element of the hash_set can be changed
    // by operating on its (non-const) reference
    Ref1 = Ref1 + 5;

    cout << "The first element in the hash_set is now "
         << *hs1.begin() << "." << endl;
}
```

```
The first element in the hash_set is 10.
The first element in the hash_set is now 15.
```

hash_set::rend

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_set (the location that had preceded the first element in the unreversed hash_set).

Remarks

`rend` is used with a reversed hash_set just as [end](#) is used with a hash_set.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the `hash_set` object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the `hash_set` object can be modified. The value returned by `rend` should not be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its `hash_set`.

Example

```
// hash_set_rend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::iterator hs1_iter;
    hash_set <int>::reverse_iterator hs1_rIter;
    hash_set <int>::const_reverse_iterator hs1_crIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_rIter = hs1.rend( );
    hs1_rIter--;
    cout << "The last element in the reversed hash_set is "
         << *hs1_rIter << "." << endl;

    // end can be used to terminate an iteration
    // through a hash_set in a forward order
    cout << "The hash_set is: ";
    for ( hs1_iter = hs1.begin( ) ; hs1_iter != hs1.end( );
          hs1_iter++ )
        cout << *hs1_iter << " ";
    cout << "." << endl;

    // rend can be used to terminate an iteration
    // through a hash_set in a reverse order
    cout << "The reversed hash_set is: ";
    for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
          hs1_rIter++ )
        cout << *hs1_rIter << " ";
    cout << "." << endl;

    hs1_rIter = hs1.rend( );
    hs1_rIter--;
    hs1.erase ( *hs1_rIter );

    hs1_rIter = hs1.rend( );
    hs1_rIter--;
    cout << "After the erasure, the last element in the "
         << "reversed hash_set is " << *hs1_rIter << "."
         << endl;
}
```

The last element in the reversed hash_set is 10.
The hash_set is: 10 20 30 .
The reversed hash_set is: 30 20 10 .
After the erasure, the last element in the reversed hash_set is 20.

hash_set::reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::reverse_iterator
reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the hash_set in reverse.

Example

See the example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

hash_set::size

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns the number of elements in the hash_set.

```
size_type size() const;
```

Return Value

The current length of the hash_set.

Remarks

Example

```
// hash_set_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1;
    hash_set<int> :: size_type i;

    hs1.insert( 1 );
    i = hs1.size( );
    cout << "The hash_set length is " << i << "." << endl;

    hs1.insert( 2 );
    i = hs1.size( );
    cout << "The hash_set length is now " << i << "." << endl;
}
```

```
The hash_set length is 1.  
The hash_set length is now 2.
```

hash_set::size_type

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

An unsigned integer type that can represent the number of elements in a hash_set.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See the example for [size](#) for an example of how to declare and use `size_type`

hash_set::swap

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Exchanges the elements of two hash_sets.

```
void swap(hash_set& right);
```

Parameters

right

The argument hash_set providing the elements to be swapped with the target hash_set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_sets whose elements are being exchanged.

Example

```

// hash_set_swap.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1, hs2, hs3;
    hash_set <int>::iterator hs1_Iter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );
    hs2.insert( 100 );
    hs2.insert( 200 );
    hs3.insert( 300 );

    cout << "The original hash_set hs1 is:";
    for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
          hs1_Iter++ )
        cout << " " << *hs1_Iter;
    cout << "." << endl;

    // This is the member function version of swap
    hs1.swap( hs2 );

    cout << "After swapping with hs2, list hs1 is:";
    for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
          hs1_Iter++ )
        cout << " " << *hs1_Iter;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( hs1, hs3 );

    cout << "After swapping with hs3, list hs1 is:";
    for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
          hs1_Iter++ )
        cout << " " << *hs1_Iter;
    cout << "." << endl;
}

```

```

The original hash_set hs1 is: 10 20 30.
After swapping with hs2, list hs1 is: 200 100.
After swapping with hs3, list hs1 is: 300.

```

hash_set::upper_bound

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Returns an iterator to the first element in a hash_set that with a key that is greater than a specified key.

```

const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);

```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

Example

```
// hash_set_upper_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_RcIter = hs1.upper_bound( 20 );
    cout << "The first element of hash_set hs1 with a key greater "
         << "than 20 is: " << *hs1_RcIter << "." << endl;

    hs1_RcIter = hs1.upper_bound( 30 );

    // If no match is found for the key, end( ) is returned
    if ( hs1_RcIter == hs1.end( ) )
        cout << "The hash_set hs1 doesn't have an element "
             << "with a key greater than 30." << endl;
    else
        cout << "The element of hash_set hs1 with a key > 40 is: "
             << *hs1_RcIter << "." << endl;

    // An element at a specific location in the hash_set can be found
    // by using a dereferenced iterator addressing the location
    hs1_AcIter = hs1.begin( );
    hs1_RcIter = hs1.upper_bound( *hs1_AcIter );
    cout << "The first element of hs1 with a key greater than "
         << endl << "that of the initial element of hs1 is: "
         << *hs1_RcIter << "." << endl;
}
```

```
The first element of hash_set hs1 with a key greater than 20 is: 30.
The hash_set hs1 doesn't have an element with a key greater than 30.
The first element of hs1 with a key greater than
that of the initial element of hs1 is: 20.
```

hash_set::value_comp

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Retrieves a copy of the comparison object used to order element values in a `hash_set`.

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a `hash_set` uses to order its elements, which is the template parameter *Compare*.

For more information on *Compare*, see the Remarks section of the [hash_set Class](#) topic.

Remarks

The stored object defines the member function:

```
bool operator( const Key& _xVal, const Key& _yVal );
```

which returns **true** if `_xVal` precedes and is not equal to `_yval` in the sort order.

Note that both [value_compare](#) and [key_compare](#) are synonyms for the template parameter *Compare*. Both types are provided for the `hash_set` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

```

// hash_set_value_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_set <int, hash_compare < int, less<int> > > hs1;
    hash_set <int, hash_compare < int, less<int> > >::value_compare
        vc1 = hs1.value_comp( );
    bool result1 = vc1( 2, 3 );
    if( result1 == true )
    {
        cout << "vc1( 2,3 ) returns value of true, "
            << "where vc1 is the function object of hs1."
            << endl;
    }
    else
    {
        cout << "vc1( 2,3 ) returns value of false, "
            << "where vc1 is the function object of hs1."
            << endl;
    }

    hash_set <int, hash_compare < int, greater<int> > > hs2;
    hash_set<int, hash_compare < int, greater<int> > >::value_compare
        vc2 = hs2.value_comp( );
    bool result2 = vc2( 2, 3 );
    if( result2 == true )
    {
        cout << "vc2( 2,3 ) returns value of true, "
            << "where vc2 is the function object of hs2."
            << endl;
    }
    else
    {
        cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of hs2."
            << endl;
    }
}

```

hash_set::value_compare

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_set to determine their relative order and a unary predicate that hashes the elements.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [hash_set Class](#) topic.

Note that both [key_compare](#) and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the `hash_set` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

See the example for [value_comp](#) for an example of how to declare and use `value_compare`.

hash_set::value_type

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

A type that describes an object stored as an element of a `hash_set` in its capacity as a value.

```
typedef Key value_type;
```

Example

```
// hash_set_value_type.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1;
    hash_set <int>::iterator hs1_Iter;

    hash_set <int> :: value_type hsvt_Int;  // Declare value_type
    hsvt_Int = 10;                        // Initialize value_type

    hash_set <int> :: key_type hskt_Int;    // Declare key_type
    hskt_Int = 20;                        // Initialize key_type

    hs1.insert( hsvt_Int );                // Insert value into hs1
    hs1.insert( hskt_Int );                // Insert key into hs1

    // A hash_set accepts key_types or value_types as elements
    cout << "The hash_set has elements:";
    for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( ); hs1_Iter++)
        cout << " " << *hs1_Iter;
    cout << "." << endl;
}
```

```
The hash_set has elements: 10 20.
```

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

hash_multiset Class

11/14/2018 • 50 minutes to read • [Edit Online](#)

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

The container class `hash_multiset` is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained serve as the key values and are not required to be unique.

Syntax

```
template <class Key, class Traits =hash_compare<Key, less<Key>>, class Allocator =allocator<Key>>
class hash_multiset
```

Parameters

Key

The element data type to be stored in the `hash_multiset`.

Traits

The type which includes two function objects, one of class `compare` that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the

`hash_compare<Key, less<Key>>` is the default value.

Allocator

The type that represents the stored allocator object that encapsulates details about the `hash_multiset`'s allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

The `hash_multiset` is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because `hash_multiset` is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency: a successful hashing performs insertions,

deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The `hash_multiset` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a `hash_multiset` may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a `hash_set` would have been the appropriate container structure. If unique definitions were attached as values to the list of unique keywords, then a `hash_map` would be an appropriate structure to contain this data. If instead the definitions were not unique, then a `hash_multimap` would be the container of choice.

The `hash_multiset` orders the sequence it controls by calling a stored hash traits object of type `value_compare`. This stored object may be accessed by calling the member function `key_comp`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values `Key` of type `Key`, the call `Trait(Key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x, y)$ is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a `hash_multiset` is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both $f(x, y)$ and $f(y, x)$ are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the `hash_multiset` class is a bidirectional iterator, but the class member functions `insert` and `hash_multiset` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own `hash_multiset` of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal `hash_multiset` of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`first`, `last`) in the context of the class member functions.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>hash_multiset</code>	Constructs a <code>hash_multiset</code> that is empty or that is a copy of all or part of some other <code>hash_multiset</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for the <code>hash_multiset</code> object.
<code>const_iterator</code>	A type that provides a bidirectional iterator that can read a const element in the <code>hash_multiset</code> .
<code>const_pointer</code>	A type that provides a pointer to a const element in a <code>hash_multiset</code> .
<code>const_reference</code>	A type that provides a reference to a const element stored in a <code>hash_multiset</code> for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in the <code>hash_multiset</code> .
<code>difference_type</code>	A signed integer type that provides the difference between two iterators that address elements within the same <code>hash_multiset</code> .
<code>iterator</code>	A type that provides a bidirectional iterator that can read or modify any element in a <code>hash_multiset</code> .
<code>key_compare</code>	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>hash_multiset</code> .
<code>key_type</code>	A type that describes an object stored as an element of a <code>hash_set</code> in its capacity as sort key.
<code>pointer</code>	A type that provides a pointer to an element in a <code>hash_multiset</code> .
<code>reference</code>	A type that provides a reference to an element stored in a <code>hash_multiset</code> .
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed <code>hash_multiset</code> .
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a <code>hash_multiset</code> .
<code>value_compare</code>	A type that provides two function objects, a binary predicate of class <code>compare</code> that can compare two element values of a <code>hash_multiset</code> to determine their relative order and a unary predicate that hashes the elements.

TYPE NAME	DESCRIPTION
<code>value_type</code>	A type that describes an object stored as an element of a <code>hash_multiset</code> in its capacity as a value.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns an iterator that addresses the first element in the <code>hash_multiset</code> .
<code>cbegin</code>	Returns a const iterator addressing the first element in the <code>hash_multiset</code> .
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a <code>hash_multiset</code> .
<code>clear</code>	Erases all the elements of a <code>hash_multiset</code> .
<code>count</code>	Returns the number of elements in a <code>hash_multiset</code> whose key matches a parameter-specified key
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed <code>hash_multiset</code> .
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed <code>hash_multiset</code> .
<code>emplace</code>	Inserts an element constructed in place into a <code>hash_multiset</code> .
<code>emplace_hint</code>	Inserts an element constructed in place into a <code>hash_multiset</code> , with a placement hint.
<code>empty</code>	Tests if a <code>hash_multiset</code> is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a <code>hash_multiset</code> .
<code>equal_range</code>	Returns a pair of iterators respectively to the first element in a <code>hash_multiset</code> with a key that is greater than a specified key and to the first element in the <code>hash_multiset</code> with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a <code>hash_multiset</code> from specified positions or removes elements that match a specified key.
<code>find</code>	Returns an iterator addressing the location of an element in a <code>hash_multiset</code> that has a key equivalent to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct the <code>hash_multiset</code> .

MEMBER FUNCTION	DESCRIPTION
insert	Inserts an element or a range of elements into a <code>hash_multiset</code> .
key_comp	Retrieves a copy of the comparison object used to order keys in a <code>hash_multiset</code> .
lower_bound	Returns an iterator to the first element in a <code>hash_multiset</code> with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the <code>hash_multiset</code> .
rbegin	Returns an iterator addressing the first element in a reversed <code>hash_multiset</code> .
rend	Returns an iterator that addresses the location succeeding the last element in a reversed <code>hash_multiset</code> .
size	Returns the number of elements in the <code>hash_multiset</code> .
swap	Exchanges the elements of two <code>hash_multiset</code> s.
upper_bound	Returns an iterator to the first element in a <code>hash_multiset</code> that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a <code>hash_multiset</code> .

Operators

OPERATOR	DESCRIPTION
hash_multiset::operator=	Replaces the elements of the <code>hash_multiset</code> with a copy of another <code>hash_multiset</code> .

Requirements

Header: `<hash_set>`

Namespace: `stdext`

`hash_multiset::allocator_type`

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that represents the allocator class for the `hash_multiset` object.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::allocator_type allocator_type;
```

Example

See example for [get_allocator](#) for an example using `allocator_type`

hash_multiset::begin

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator that addresses the first element in the hash_multiset.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_multiset or the location succeeding an empty hash_multiset.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_multiset object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_multiset object can be modified.

Example

```
// hash_multiset_begin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int>::iterator hms1_Iter;
    hash_multiset <int>::const_iterator hms1_cIter;

    hms1.insert( 1 );
    hms1.insert( 2 );
    hms1.insert( 3 );

    hms1_Iter = hms1.begin( );
    cout << "The first element of hms1 is " << *hms1_Iter << endl;

    hms1_Iter = hms1.begin( );
    hms1.erase( hms1_Iter );

    // The following 2 lines would err because the iterator is const
    // hms1_cIter = hms1.begin( );
    // hms1.erase( hms1_cIter );

    hms1_cIter = hms1.begin( );
    cout << "The first element of hms1 is now " << *hms1_cIter << endl;
}
```

```
The first element of hms1 is 1
The first element of hms1 is now 2
```

hash_multiset::cbegin

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a const iterator that addresses the first element in the hash_multiset.

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the [hash_multiset](#) or the location succeeding an empty `hash_multiset`.

Remarks

With the return value of `cbegin`, the elements in the `hash_multiset` object cannot be modified.

Example

```
// hash_multiset_cbegin.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hs1;
    hash_multiset <int>::const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_cIter = hs1.cbegin( );
    cout << "The first element of hs1 is " << *hs1_cIter << endl;
}
```

```
The first element of hs1 is 1
```

hash_multiset::cend

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a const iterator that addresses the location succeeding the last element in a hash_multiset.


```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a [hash_multiset](#). If the `hash_multiset` is empty, then `hash_multiset::cend == hash_multiset::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_multiset`. The value returned by `cend` should not be dereferenced.

Example

```
// hash_multiset_cend.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hs1;
    hash_multiset <int> :: const_iterator hs1_cIter;

    hs1.insert( 1 );
    hs1.insert( 2 );
    hs1.insert( 3 );

    hs1_cIter = hs1.cend( );
    hs1_cIter--;
    cout << "The last element of hs1 is " << *hs1_cIter << endl;
}
```

```
The last element of hs1 is 3
```

hash_multiset::clear

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Erases all the elements of a `hash_multiset`.

```
void clear();
```

Remarks

Example

```
// hash_multiset_clear.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;

    hms1.insert( 1 );
    hms1.insert( 2 );

    cout << "The size of the hash_multiset is initially " << hms1.size( )
        << "." << endl;

    hms1.clear( );
    cout << "The size of the hash_multiset after clearing is "
        << hms1.size( ) << "." << endl;
}
```

```
The size of the hash_multiset is initially 2.
The size of the hash_multiset after clearing is 0.
```

hash_multiset::const_iterator

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a bidirectional iterator that can read a **const** element in the hash_multiset.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See example for [begin](#) for an example using `const_iterator`.

hash_multiset::const_pointer

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a pointer to a **const** element in a hash_multiset.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, a [const_iterator](#) should be used to access the elements in a **const** hash_multiset object.

hash_multiset::const_reference

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a reference to a **const** element stored in a hash_multiset for reading and performing **const** operations.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::const_reference
const_reference;
```

Remarks

Example

```
// hash_multiset_const_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;

    hms1.insert( 10 );
    hms1.insert( 20 );

    // Declare and initialize a const_reference &Ref1
    // to the 1st element
    const int &Ref1 = *hms1.begin( );

    cout << "The first element in the hash_multiset is "
         << Ref1 << "." << endl;

    // The following line would cause an error because the
    // const_reference cannot be used to modify the hash_multiset
    // Ref1 = Ref1 + 5;
}
```

```
The first element in the hash_multiset is 10.
```

hash_multiset::const_reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a bidirectional iterator that can read any **const** element in the hash_multiset.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::const_reverse_iterator
const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the `hash_multiset` in reverse.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`.

hash_multiset::count

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns the number of elements in a `hash_multiset` whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the `hash_multiset`.

Return Value

The number of elements in the `hash_multiset` with the parameter-specified key.

Remarks

The member function returns the number of elements in the following range:

[`lower_bound(key)`, `upper_bound(key)`).

Example

The following example demonstrates the use of the `hash_multiset::count` member function.

```

// hash_multiset_count.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1;
    hash_multiset<int>::size_type i;

    hms1.insert(1);
    hms1.insert(1);

    // Keys do not need to be unique in hash_multiset,
    // so duplicates may exist.
    i = hms1.count(1);
    cout << "The number of elements in hms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hms1.count(2);
    cout << "The number of elements in hms1 with a sort key of 2 is: "
         << i << "." << endl;
}

```

```

The number of elements in hms1 with a sort key of 1 is: 2.
The number of elements in hms1 with a sort key of 2 is: 0.

```

hash_multiset::crbegin

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a const iterator addressing the first element in a reversed hash_multiset.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [hash_multiset](#) or addressing what had been the last element in the unreversed `hash_multiset`.

Remarks

`crbegin` is used with a reversed `hash_multiset` just as [hash_multiset::begin](#) is used with a `hash_multiset`.

With the return value of `crbegin`, the `hash_multiset` object cannot be modified.

`crbegin` can be used to iterate through a `hash_multiset` backwards.

Example

```
// hash_multiset_crbegin.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hs1;
    hash_multiset <int>::const_reverse_iterator hs1_crIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_crIter = hs1.crbegin( );
    cout << "The first element in the reversed hash_multiset is "
         << *hs1_crIter << "." << endl;
}
```

The first element in the reversed hash_multiset is 30.

hash_multiset::crend

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_multiset.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [hash_multiset](#) (the location that had preceded the first element in the unreversed `hash_multiset`).

Remarks

`crend` is used with a reversed `hash_multiset` just as [hash_multiset::end](#) is used with a `hash_multiset`.

With the return value of `crend`, the `hash_multiset` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its hash_multiset.

Example

```
// hash_multiset_crend.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hs1;
    hash_multiset <int>::const_reverse_iterator hs1_crIter;

    hs1.insert( 10 );
    hs1.insert( 20 );
    hs1.insert( 30 );

    hs1_crIter = hs1.crend( );
    hs1_crIter--;
    cout << "The last element in the reversed hash_multiset is "
         << *hs1_crIter << "." << endl;
}
```

The last element in the reversed hash_multiset is 10.

hash_multiset::difference_type

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A signed integer type that provides the difference between two iterators that address elements within the same hash_multiset.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::difference_type
difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container.

The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set. Subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector or deque.

Example

```

// hash_multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_multiset <int> hms1;
    hash_multiset <int>::iterator hms1_iter, hms1_bIter, hms1_eIter;

    hms1.insert( 20 );
    hms1.insert( 10 );

    // hash_multiset elements need not be unique
    hms1.insert( 20 );

    hms1_bIter = hms1.begin( );
    hms1_eIter = hms1.end( );

    hash_multiset <int>::difference_type df_typ5, df_typ10,
        df_typ20;

    df_typ5 = count( hms1_bIter, hms1_eIter, 5 );
    df_typ10 = count( hms1_bIter, hms1_eIter, 10 );
    df_typ20 = count( hms1_bIter, hms1_eIter, 20 );

    // The keys & hence the elements of a hash_multiset
    // need not be unique and may occur multiple times
    cout << "The number '5' occurs " << df_typ5
        << " times in hash_multiset hms1.\n";
    cout << "The number '10' occurs " << df_typ10
        << " times in hash_multiset hms1.\n";
    cout << "The number '20' occurs " << df_typ20
        << " times in hash_multiset hms1.\n";

    // Count the number of elements in a hash_multiset
    hash_multiset <int>::difference_type df_count = 0;
    hms1_iter = hms1.begin( );
    while ( hms1_iter != hms1_eIter)
    {
        df_count++;
        hms1_iter++;
    }

    cout << "The number of elements in the hash_multiset hms1 is "
        << df_count << "." << endl;
}

```

The number '5' occurs 0 times in hash_multiset hms1.
 The number '10' occurs 1 times in hash_multiset hms1.
 The number '20' occurs 2 times in hash_multiset hms1.
 The number of elements in the hash_multiset hms1 is 3.

hash_multiset::emplace

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Inserts an element constructed in place into a `hash_multiset`.

```
template <class ValTy>
iterator insert(ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of an element to be inserted into the hash_multiset unless the <code>hash_multiset</code> already contains that element or, more generally, an element whose key is equivalently ordered.

Return Value

The `emplace` member function returns an iterator that points to the position where the new element was inserted.

Remarks

Example

```
// hash_multiset_emplace.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<string> hms3;
    string str1("a");

    hms3.emplace(move(str1));
    cout << "After the emplace insertion, hms3 contains "
         << *hms3.begin() << "." << endl;
}
```

After the `emplace` insertion, `hms3` contains `a`.

hash_multiset::emplace_hint

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Inserts an element constructed in place into a `hash_multiset`, with a placement hint.

```
template <class ValTy>
iterator insert(
    const_iterator _where,
    ValTy&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The value of an element to be inserted into the hash_multiset unless the <code>hash_multiset</code> already contains that element or, more generally, an element whose key is equivalently ordered.
<i>_Where</i>	The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows <i>_Where</i> .)

Return Value

The [hash_multiset::emplace](#) member function returns an iterator that points to the position where the new element was inserted into the `hash_multiset`.

Remarks

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

```
// hash_multiset_emplace_hint.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<string> hms1;
    string str1("a");

    hms1.insert(hms1.begin(), move(str1));
    cout << "After the emplace insertion, hms1 contains "
         << *hms1.begin() << "." << endl;
}
```

After the emplace insertion, hms1 contains a.

hash_multiset::empty

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Tests if a `hash_multiset` is empty.

```
bool empty() const;
```

Return Value

true if the hash_multiset is empty; **false** if the hash_multiset is nonempty.

Remarks

Example

```
// hash_multiset_empty.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1, hms2;
    hms1.insert ( 1 );

    if ( hms1.empty( ) )
        cout << "The hash_multiset hms1 is empty." << endl;
    else
        cout << "The hash_multiset hms1 is not empty." << endl;

    if ( hms2.empty( ) )
        cout << "The hash_multiset hms2 is empty." << endl;
    else
        cout << "The hash_multiset hms2 is not empty." << endl;
}
```

```
The hash_multiset hms1 is not empty.
The hash_multiset hms2 is empty.
```

hash_multiset::end

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator that addresses the location succeeding the last element in a hash_multiset.

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multiset. If the hash_multiset is empty, then hash_multiset::end == hash_multiset::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_multiset. The value returned by `end` should not be dereferenced.

Example

```

// hash_multiset_end.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int> :: iterator hms1_Iter;
    hash_multiset <int> :: const_iterator hms1_cIter;

    hms1.insert( 1 );
    hms1.insert( 2 );
    hms1.insert( 3 );

    hms1_Iter = hms1.end( );
    hms1_Iter--;
    cout << "The last element of hms1 is " << *hms1_Iter << endl;

    hms1.erase( hms1_Iter );

    // The following 3 lines would err because the iterator is const
    // hms1_cIter = hms1.end( );
    // hms1_cIter--;
    // hms1.erase( hms1_cIter );

    hms1_cIter = hms1.end( );
    hms1_cIter--;
    cout << "The last element of hms1 is now " << *hms1_cIter << endl;
}

```

```

The last element of hms1 is 3
The last element of hms1 is now 2

```

hash_multiset::equal_range

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a pair of iterators respectively to the first element in a hash_multiset with a key that is greater than a specified key and to the first element in the hash_multiset with a key that is equal to or greater than the key.

```

pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);

```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

A pair of iterators where the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the

lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```
// hash_multiset_equal_range.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    typedef hash_multiset<int> IntHSet;
    IntHSet hms1;
    hash_multiset <int> :: const_iterator hms1_RcIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
    p1 = hms1.equal_range( 20 );

    cout << "The upper bound of the element with "
         << "a key of 20\nin the hash_multiset hms1 is: "
         << *(p1.second) << "." << endl;

    cout << "The lower bound of the element with "
         << "a key of 20\nin the hash_multiset hms1 is: "
         << *(p1.first) << "." << endl;

    // Compare the upper_bound called directly
    hms1_RcIter = hms1.upper_bound( 20 );
    cout << "A direct call of upper_bound( 20 ) gives "
         << *hms1_RcIter << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

    p2 = hms1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == hms1.end( ) )
        && ( p2.second == hms1.end( ) ) )
        cout << "The hash_multiset hms1 doesn't have an element "
             << "with a key less than 40." << endl;
    else
        cout << "The element of hash_multiset hms1"
             << "with a key >= 40 is: "
             << *(p1.first) << "." << endl;
}
```

The upper bound of the element with a key of 20
in the hash_multiset hms1 is: 30.
The lower bound of the element with a key of 20
in the hash_multiset hms1 is: 20.
A direct call of upper_bound(20) gives 30,
matching the 2nd element of the pair returned by equal_range(20).
The hash_multiset hms1 doesn't have an element with a key less than 40.

hash_multiset::erase

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Removes an element or a range of elements in a `hash_multiset` from specified positions or removes elements that match a specified key.

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

_Where

Position of the element to be removed from the `hash_multiset`.

first

Position of the first element removed from the `hash_multiset`.

last

Position just beyond the last element removed from the `hash_multiset`.

key

The key of the elements to be removed from the `hash_multiset`.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the `hash_multiset` if no such element exists. For the third member function, the number of elements that have been removed from the `hash_multiset`.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the `hash_multiset::erase` member function.

```
// hash_multiset_erase.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1, hms2, hms3;
    hash_multiset<int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_multiset<int>::size_type n;

    for (i = 1; i < 5; i++)
    {
        hms1.insert(i);
        hms2.insert(i * i);
        hms3.insert(i - 1);
    }

    // The 1st member function removes an element at a given position
```

```

Iter1 = ++hms1.begin();
hms1.erase(Iter1);

cout << "After the 2nd element is deleted,\n"
      << "the hash_multiset hms1 is:" ;
for (pIter = hms1.begin(); pIter != hms1.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 2nd member function removes elements
// in the range [ first, last)
Iter1 = ++hms2.begin();
Iter2 = --hms2.end();
hms2.erase(Iter1, Iter2);

cout << "After the middle two elements are deleted,\n"
      << "the hash_multiset hms2 is:" ;
for (pIter = hms2.begin(); pIter != hms2.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 3rd member function removes elements with a given key
n = hms3.erase(2);

cout << "After the element with a key of 2 is deleted,\n"
      << "the hash_multiset hms3 is:" ;
for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;

// The 3rd member function returns the number of elements removed
cout << "The number of elements removed from hms3 is: "
      << n << "." << endl;

// The dereferenced iterator can also be used to specify a key
Iter1 = ++hms3.begin();
hms3.erase(Iter1);

cout << "After another element with a key "
      << "equal to that of the 2nd element\n"
      << "is deleted, the hash_multiset hms3 is:" ;
for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
    cout << " " << *pIter;
cout << "." << endl;
}

```

```

After the 2nd element is deleted,
the hash_multiset hms1 is: 1 3 4.
After the middle two elements are deleted,
the hash_multiset hms2 is: 16 4.
After the element with a key of 2 is deleted,
the hash_multiset hms3 is: 0 1 3.
The number of elements removed from hms3 is: 1.
After another element with a key equal to that of the 2nd element
is deleted, the hash_multiset hms3 is: 0 3.

```

hash_multiset::find

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator addressing the location of an element in a hash_multiset that has a key equivalent to a

specified key.

```
iterator find(const Key& key);  
  
const_iterator find(const Key& key) const;
```

Parameters

key

The argument key to be matched by the sort key of an element from the hash_multiset being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multiset whose sort key is [equivalent](#) to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of [find](#) is assigned to a [const_iterator](#), the hash_multiset object cannot be modified. If the return value of [find](#) is assigned to an [iterator](#), the hash_multiset object can be modified.

Example


```

// hash_multiset_find.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    hms1_RcIter = hms1.find( 20 );
    cout << "The element of hash_multiset hms1 with a key of 20 is: "
         << *hms1_RcIter << "." << endl;

    hms1_RcIter = hms1.find( 40 );

    // If no match is found for the key, end( ) is returned
    if ( hms1_RcIter == hms1.end( ) )
        cout << "The hash_multiset hms1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of hash_multiset hms1 with a key of 40 is: "
             << *hms1_RcIter << "." << endl;

    // The element at a specific location in the hash_multiset can be found
    // by using a dereferenced iterator addressing the location
    hms1_AcIter = hms1.end( );
    hms1_AcIter--;
    hms1_RcIter = hms1.find( *hms1_AcIter );
    cout << "The element of hms1 with a key matching "
         << "that of the last element is: "
         << *hms1_RcIter << "." << endl;
}

```

```

The element of hash_multiset hms1 with a key of 20 is: 20.
The hash_multiset hms1 doesn't have an element with a key of 40.
The element of hms1 with a key matching that of the last element is: 30.

```

hash_multiset::get_allocator

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns a copy of the allocator object used to construct the hash_multiset.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_multiset to manage memory, which is the class's template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the [hash_multiset Class](#) topic.

Remarks

Allocators for the `hash_multiset` class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// hash_multiset_get_allocator.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    // The following lines declare objects
    // that use the default allocator.
    hash_multiset <int, hash_compare <int, less<int> > > hms1;
    hash_multiset <int, hash_compare <int, greater<int> > > hms2;
    hash_multiset <double, hash_compare <double,
        less<double> >, allocator<double> > hms3;

    hash_multiset <int, hash_compare <int,
        greater<int> > >::allocator_type hms2_Alloc;
    hash_multiset <double>::allocator_type hms3_Alloc;
    hms2_Alloc = hms2.get_allocator( );

    cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms1.max_size( ) << "." << endl;

    cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms3.max_size( ) << "." << endl;

    // The following lines create a hash_multiset hms4
    // with the allocator of hash_multiset hms1.
    hash_multiset <int>::allocator_type hms4_Alloc;
    hash_multiset <int> hms4;
    hms4_Alloc = hms2.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated by the other
    if( hms2_Alloc == hms4_Alloc )
    {
        cout << "The allocators are interchangeable."
            << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
            << endl;
    }
}
```

`hash_multiset::hash_multiset`

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Constructs a `hash_multiset` that is empty or that is a copy of all or part of some other `hash_multiset`.

```
hash_multiset();

explicit hash_multiset(
    const Traits& Comp);

hash_multiset(
    const Traits& Comp,
    const Allocator& Al);

hash_multiset(
    const hash_multiset<Key, Traits, Allocator>& Right);

hash_multiset(
    hash_multiset&& Right
);
hash_multiset (initializer_list<Type> IList);

hash_multiset(
    initializer_list<Tu[e> IList, const Compare& Comp):
hash_multiset(
    initializer_list<Type> IList, const Compare& Comp, const Allocator& Al);

template <class InputIterator>
hash_multiset(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_multiset(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
hash_multiset(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this <code>hash_multiset</code> object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the <code>hash_multiset</code> , which defaults to <code>hash_compare</code> .
<i>Right</i>	The <code>hash_multiset</code> of which the constructed <code>hash_multiset</code> is to be a copy.

PARAMETER	DESCRIPTION
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.
<i>lList</i>	The initializer_list that contains the elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the `hash_multiset` and that can later be returned by calling `hash_multiset::get_allocator`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their `hash_multisets`.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_multiset` and that can later be returned by calling `hash_multiset::key_comp`. For more information on `Traits` see the [hash_multiset Class](#) topic.

The first three constructors specify an empty initial `hash_multiset`, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor moves the `hash_multiset` `Right`.

The fifth, sixth, and seventh constructors use an `initializer_list`.

The last three constructors copy the range [`First`, `Last`) of a `hash_multiset` with increasing explicitness in specifying the type of comparison function of class `Compare` and allocator.

The actual order of elements in a hashed set container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

hash_multiset::insert

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Inserts an element or a range of elements into a `hash_multiset`.

```

iterator insert(
    const Type& Val);

iterator insert(
    iterator Where,
    const Type& Al);

void insert(
    initializer_list<Type> IList);

iterator insert(
    const Type& Val);

iterator insert(
    Iterator Where,
    const Type& Val);

template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

template <class ValTy>
iterator insert(
    ValTy&& Val);

template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the hash_multiset unless the hash_multiset already contains that element or, more generally, an element whose key is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows <code>_where</code> .)
<i>First</i>	The position of the first element to be copied from a hash_multiset.
<i>Last</i>	The position just beyond the last element to be copied from a hash_multiset.
<i>IList</i>	The initializer_list that contains the elements to copy.

Return Value

The first two insert member functions return an iterator that points to the position where the new element was inserted.

The next three member functions use an initializer_list.

The third member function inserts the sequence of element values into a hash_multiset corresponding to each element addressed by an iterator of in the range [`First` , `Last`) of a specified hash_multiset.

Remarks

Insertion can occur in amortized constant time for the hint version of `insert`, instead of logarithmic time, if the insertion point immediately follows *Where*.

hash_multiset::iterator

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a bidirectional iterator that can read or modify any element in a `hash_multiset`.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::iterator iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See example for [begin](#) for an example of how to declare and use `iterator`.

hash_multiset::key_comp

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Retrieves a copy of the comparison object used to order keys in a `hash_multiset`.

```
key_compare key_comp() const;
```

Return Value

Returns the `hash_multiset` template parameter *Traits*, which contains function objects that are used to hash and to order the elements of the container.

For more information on *Traits* see the [hash_multiset Class](#) topic.

Remarks

The stored object defines a member function:

```
bool operator( const Key& _xVal, const Key& _yVal );
```

which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both [key_compare](#) and [value_compare](#) are synonyms for the template parameter *Traits*. Both types are provided for the `hash_multiset` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

```

// hash_multiset_key_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_multiset <int, hash_compare < int, less<int> > >hms1;
    hash_multiset<int, hash_compare < int, less<int> > >::key_compare kc1
        = hms1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
            << "where kc1 is the function object of hms1."
            << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
            << "where kc1 is the function object of hms1."
            << endl;
    }

    hash_multiset <int, hash_compare < int, greater<int> > > hms2;
    hash_multiset<int, hash_compare < int, greater<int> > >::key_compare
        kc2 = hms2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true, "
            << "where kc2 is the function object of hms2."
            << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
            << "where kc2 is the function object of hms2."
            << endl;
    }
}

```

hash_multiset::key_compare

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [hash_multiset Class](#) topic.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the `hash_set` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

See example for [key_comp](#) for an example of how to declare and use `key_compare`.

hash_multiset::key_type

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the `hash_multiset`.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

Note that both `key_type` and `value_type` are synonyms for the template parameter *Key*. Both types are provided for the `set` and `multiset` classes, where they are identical, for compatibility with the `map` and `multimap` classes, where they are distinct.

For more information on *Key*, see the Remarks section of the [hash_multiset Class](#) topic.

Example

See example for [value_type](#) for an example of how to declare and use `key_type`.

hash_multiset::lower_bound

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator to the first element in a `hash_multiset` with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(const Key& key) const;  
  
iterator lower_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the `hash_multiset` being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of the first element in a `hash_multiset` with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the `hash_multiset` if no match is found for the key.

Remarks

Example

```
// hash_multiset_lower_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main() {
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    hms1_RcIter = hms1.lower_bound( 20 );
    cout << "The element of hash_multiset hms1 with a key of 20 is: "
         << *hms1_RcIter << "." << endl;

    hms1_RcIter = hms1.lower_bound( 40 );

    // If no match is found for the key, end( ) is returned
    if ( hms1_RcIter == hms1.end( ) )
        cout << "The hash_multiset hms1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of hash_multiset hms1 with a key of 40 is: "
             << *hms1_RcIter << "." << endl;

    // An element at a specific location in the hash_multiset can be found
    // by using a dereferenced iterator that addresses the location
    hms1_AcIter = hms1.end( );
    hms1_AcIter--;
    hms1_RcIter = hms1.lower_bound( *hms1_AcIter );
    cout << "The element of hms1 with a key matching "
         << "that of the last element is: "
         << *hms1_RcIter << "." << endl;
}
```

hash_multiset::max_size

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns the maximum length of the hash_multiset.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_multiset.

Remarks

Example

```

// hash_multiset_max_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int>::size_type i;

    i = hms1.max_size( );
    cout << "The maximum possible length "
         << "of the hash_multiset is " << i << "." << endl;
}

```

hash_multiset::operator=

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Replaces the elements of the hash_multiset with a copy of another hash_multiset.

```

hash_multiset& operator=(const hash_multiset& right);

hash_multiset& operator=(hash_multiset&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The hash_multiset being copied into the <code>hash_multiset</code> .

Remarks

After erasing any existing elements in a `hash_multiset`, `operator=` either copies or moves the contents of *right* into the `hash_multiset`.

Example

```

// hash_multiset_operator_as.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> v1, v2, v3;
    hash_multiset<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter << " ";
    cout << endl;
}

```

hash_multiset::pointer

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a pointer to an element in a hash_multiset.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a multiset object.

hash_multiset::rbegin

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator addressing the first element in a reversed hash_multiset.

```
const_reverse_iterator rbegin() const;
```

```
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multiset or addressing what had been the last element in the unreversed hash_multiset.

Remarks

`rbegin` is used with a reversed hash_multiset just as `begin` is used with a hash_multiset.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_multiset object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_multiset object can be modified.

`rbegin` can be used to iterate through a hash_multiset backwards.

Example

```

// hash_multiset_rbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int>::iterator hms1_Iter;
    hash_multiset <int>::reverse_iterator hms1_rIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    hms1_rIter = hms1.rbegin( );
    cout << "The first element in the reversed hash_multiset is "
         << *hms1_rIter << "." << endl;

    // begin can be used to start an iteration
    // through a hash_multiset in a forward order
    cout << "The hash_multiset is: ";
    for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( ) ;
          hms1_Iter++ )
        cout << *hms1_Iter << " ";
    cout << endl;

    // rbegin can be used to start an iteration
    // through a hash_multiset in a reverse order
    cout << "The reversed hash_multiset is: ";
    for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( ) ;
          hms1_rIter++ )
        cout << *hms1_rIter << " ";
    cout << endl;

    // A hash_multiset element can be erased by dereferencing to its key
    hms1_rIter = hms1.rbegin( );
    hms1.erase ( *hms1_rIter );

    hms1_rIter = hms1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed hash_multiset is "<< *hms1_rIter << "."
         << endl;
}

```

```

The first element in the reversed hash_multiset is 30.
The hash_multiset is: 10 20 30
The reversed hash_multiset is: 30 20 10
After the erasure, the first element in the reversed hash_multiset is 20.

```

hash_multiset::reference

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a reference to an element stored in a hash_multiset.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::reference reference;
```

Remarks

Example

```
// hash_multiset_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;

    hms1.insert( 10 );
    hms1.insert( 20 );

    // Declare and initialize a reference &Ref1 to the 1st element
    int &Ref1 = *hms1.begin( );

    cout << "The first element in the hash_multiset is "
         << Ref1 << "." << endl;

    // The value of the 1st element of the hash_multiset can be
    // changed by operating on its (non const) reference
    Ref1 = Ref1 + 5;

    cout << "The first element in the hash_multiset is now "
         << *hms1.begin() << "." << endl;
}
```

```
The first element in the hash_multiset is 10.
The first element in the hash_multiset is now 15.
```

hash_multiset::rend

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multiset.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multiset (the location that had preceded the first element in the unreversed hash_multiset).

Remarks

`rend` is used with a reversed hash_multiset just as `end` is used with a hash_multiset.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the `hash_multiset` object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the `hash_multiset` object can be modified. The value returned by `rend` should not be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its `hash_multiset`.

Example

```
// hash_multiset_rend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int>::iterator hms1_Iter;
    hash_multiset <int>::reverse_iterator hms1_rIter;
    hash_multiset <int>::const_reverse_iterator hms1_crIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    hms1_rIter = hms1.rend( );
    hms1_rIter--;
    cout << "The last element in the reversed hash_multiset is "
         << *hms1_rIter << "." << endl;

    // end can be used to terminate an iteration
    // through a hash_multiset in a forward order
    cout << "The hash_multiset is: ";
    for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( ) ;
          hms1_Iter++ )
        cout << *hms1_Iter << " ";
    cout << "." << endl;

    // rend can be used to terminate an iteration
    // through a hash_multiset in a reverse order
    cout << "The reversed hash_multiset is: ";
    for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( ) ;
          hms1_rIter++ )
        cout << *hms1_rIter << " ";
    cout << "." << endl;

    hms1_rIter = hms1.rend( );
    hms1_rIter--;
    hms1.erase ( *hms1_rIter );

    hms1_rIter = hms1.rend( );
    hms1_rIter--;
    cout << "After the erasure, the last element in the "
         << "reversed hash_multiset is " << *hms1_rIter << "."
         << endl;
}
```

The last element in the reversed hash_multiset is 10.
The hash_multiset is: 10 20 30 .
The reversed hash_multiset is: 30 20 10 .
After the erasure, the last element in the reversed hash_multiset is 20.

hash_multiset::reverse_iterator

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multiset.

```
typedef list<typename Traits::value_type, typename Traits::allocator_type>::reverse_iterator
reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the hash_multiset in reverse.

Example

See example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

hash_multiset::size

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns the number of elements in the hash_multiset.

```
size_type size() const;
```

Return Value

The current length of the hash_multiset.

Remarks

Example

```
// hash_multiset_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int> :: size_type i;

    hms1.insert( 1 );
    i = hms1.size( );
    cout << "The hash_multiset length is " << i << "." << endl;

    hms1.insert( 2 );
    i = hms1.size( );
    cout << "The hash_multiset length is now " << i << "." << endl;
}
```



```
The hash_multiset length is 1.  
The hash_multiset length is now 2.
```

hash_multiset::size_type

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

An unsigned integer type that can represent the number of elements in a hash_multiset.

```
typedef list<typename _Traits::value_type, typename _Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See example for [size](#) for an example of how to declare and use `size_type`

hash_multiset::swap

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Exchanges the elements of two hash_multisets.

```
void swap(hash_multiset& right);
```

Parameters

right

The argument hash_multiset providing the elements to be swapped with the target hash_multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multisets whose elements are being exchanged.

Example

```

// hash_multiset_swap.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1, hms2, hms3;
    hash_multiset <int>::iterator hms1_Iter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );
    hms2.insert( 100 );
    hms2.insert( 200 );
    hms3.insert( 300 );

    cout << "The original hash_multiset hms1 is:";
    for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
          hms1_Iter++ )
        cout << " " << *hms1_Iter;
    cout << "." << endl;

    // This is the member function version of swap
    hms1.swap( hms2 );

    cout << "After swapping with hms2, list hms1 is:";
    for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
          hms1_Iter++ )
        cout << " " << *hms1_Iter;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( hms1, hms3 );

    cout << "After swapping with hms3, list hms1 is:";
    for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
          hms1_Iter++ )
        cout << " " << *hms1_Iter;
    cout << "." << endl;
}

```

The original hash_multiset hms1 is: 10 20 30.
 After swapping with hms2, list hms1 is: 200 100.
 After swapping with hms3, list hms1 is: 300.

hash_multiset::upper_bound

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Returns an iterator to the first element in a hash_multiset with a key that is greater than a specified key.

```

const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);

```

Parameters

key

The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

An [iterator](#) or [const_iterator](#) that addresses the location of the first element in a hash_multiset with a key greater than the argument key, or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

Example

```
// hash_multiset_upper_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

    hms1.insert( 10 );
    hms1.insert( 20 );
    hms1.insert( 30 );

    hms1_RcIter = hms1.upper_bound( 20 );
    cout << "The first element of hash_multiset hms1" << endl
         << "with a key greater than 20 is: "
         << *hms1_RcIter << "." << endl;

    hms1_RcIter = hms1.upper_bound( 30 );

    // If no match is found for the key, end( ) is returned
    if ( hms1_RcIter == hms1.end( ) )
        cout << "The hash_multiset hms1 doesn't have an element\n"
             << "with a key greater than 30." << endl;
    else
        cout << "The element of hash_multiset hms1"
             << "with a key > 40 is: "
             << *hms1_RcIter << "." << endl;

    // An element at a specific location in the hash_multiset can be
    // found by using a dereferenced iterator addressing the location
    hms1_AcIter = hms1.begin( );
    hms1_RcIter = hms1.upper_bound( *hms1_AcIter );
    cout << "The first element of hms1 with a key greater than "
         << endl << "that of the initial element of hms1 is: "
         << *hms1_RcIter << "." << endl;
}
```

```
The first element of hash_multiset hms1
with a key greater than 20 is: 30.
The hash_multiset hms1 doesn't have an element
with a key greater than 30.
The first element of hms1 with a key greater than
that of the initial element of hms1 is: 20.
```

hash_multiset::value_comp

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

Retrieves a copy of the comparison object used to order element values in a `hash_multiset`.

```
value_compare value_comp() const;
```

Return Value

Returns the `hash_multiset` template parameter *Traits*, which contains function objects that are used to hash and to order elements of the container.

For more information on *Traits* see the [hash_multiset Class](#) topic.

Remarks

The stored object defines a member function:

```
bool operator( constKey& _xVal , const Key& _yVal);
```

which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both [key_compare](#) and [value_compare](#) are synonyms for the template parameter *Traits*. Both types are provided for the `hash_multiset` and `hash_multiset` classes, where they are identical, for compatibility with the `hash_map` and `hash_multimap` classes, where they are distinct.

Example

```

// hash_multiset_value_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;

    hash_multiset <int, hash_compare < int, less<int> > > hms1;
    hash_multiset <int, hash_compare < int, less<int> > >::value_compare
        vc1 = hms1.value_comp( );
    bool result1 = vc1( 2, 3 );
    if( result1 == true )
    {
        cout << "vc1( 2,3 ) returns value of true, "
            << "where vc1 is the function object of hms1."
            << endl;
    }
    else
    {
        cout << "vc1( 2,3 ) returns value of false, "
            << "where vc1 is the function object of hms1."
            << endl;
    }

    hash_multiset <int, hash_compare < int, greater<int> > > hms2;
    hash_multiset<int, hash_compare < int, greater<int> > >::
        value_compare vc2 = hms2.value_comp( );
    bool result2 = vc2( 2, 3 );
    if( result2 == true )
    {
        cout << "vc2( 2,3 ) returns value of true, "
            << "where vc2 is the function object of hms2."
            << endl;
    }
    else
    {
        cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of hms2."
            << endl;
    }
}

```

```

vc1( 2,3 ) returns value of true, where vc1 is the function object of hms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of hms2.

```

hash_multiset::value_compare

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [hash_multiset Class](#) topic.

Note that both [key_compare](#) and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the classes `set` and `multiset`, where they are identical, for compatibility with the classes `map` and `multimap`, where they are distinct.

Example

See example for [value_comp](#) for an example of how to declare and use `value_compare`.

hash_multiset::value_type

NOTE

This API is obsolete. The alternative is [unordered_multiset Class](#).

A type that describes an object stored as an element as a `hash_multiset` in its capacity as a value.

```
typedef Key value_type;
```

Example

```
// hash_multiset_value_type.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hms1;
    hash_multiset <int>::iterator hms1_Iter;

    // Declare value_type
    hash_multiset <int> :: value_type hmsvt_Int;

    hmsvt_Int = 10;    // Initialize value_type

    // Declare key_type
    hash_multiset <int> :: key_type hmskt_Int;
    hmskt_Int = 20;    // Initialize key_type

    hms1.insert( hmsvt_Int );    // Insert value into s1
    hms1.insert( hmskt_Int );    // Insert key into s1

    // A hash_multiset accepts key_types or value_types as elements
    cout << "The hash_multiset has elements:";
    for ( hms1_Iter = hms1.begin() ; hms1_Iter != hms1.end( );
          hms1_Iter++)
        cout << " " << *hms1_Iter;
    cout << "." << endl;
}
```

```
The hash_multiset has elements: 10 20.
```

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<hash_set> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

swap (hash_multiset)

swap

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Exchanges the elements of two hash_sets.

```
void swap(  
    hash_set <Key, Traits, Allocator>& left,  
    hash_set <Key, Traits, Allocator>& right);
```

Parameters

right

The hash_set providing the elements to be swapped, or the hash_set whose elements are to be exchanged with those of the hash_set *left*.

left

The hash_set whose elements are to be exchanged with those of the hash_set *right*.

Remarks

The `swap` template function is an algorithm specialized on the container class `hash_set` to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class [hash_set::swap](#) for an example that uses the template version of `swap`.

swap (hash_multiset)

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Exchanges the elements of two hash_multisets.


```
void swap(hash_multiset <Key, Traits, Allocator>& left, hash_multiset <Key, Traits, Allocator>& right);
```

Parameters

right

The hash_multiset providing the elements to be swapped, or the hash_multiset whose elements are to be exchanged with those of the hash_multiset *left*.

left

The hash_multiset whose elements are to be exchanged with those of the hash_multiset *right*.

Remarks

The `swap` template function is an algorithm specialized on the container class `hash_multiset` to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class `hash_multiset::swap` for an example that uses the template version of `swap`.

See also

[<hash_set>](#)

<hash_set> operators

3/28/2019 • 4 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator!= (hash_multiset)</code>	<code>operator==</code>
<code>operator== (hash_multiset)</code>		

operator!=

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Tests if the `hash_set` object on the left side of the operator is not equal to the `hash_set` object on the right side.

```
bool operator!=(const hash_set <Key, Traits, Allocator>& left, const hash_set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `hash_set`.

right

An object of type `hash_set`.

Return Value

true if the `hash_sets` are not equal; **false** if `hash_sets` are equal.

Remarks

The comparison between `hash_set` objects is based on a pairwise comparison between their elements. Two `hash_sets` are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Members of the `<hash_map>` and `<hash_set>` header files are in the [stdext Namespace](#).

Example

```

// hash_set_op_ne.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> hs1, hs2, hs3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        hs1.insert ( i );
        hs2.insert ( i * i );
        hs3.insert ( i );
    }

    if ( hs1 != hs2 )
        cout << "The hash_sets hs1 and hs2 are not equal." << endl;
    else
        cout << "The hash_sets hs1 and hs2 are equal." << endl;

    if ( hs1 != hs3 )
        cout << "The hash_sets hs1 and hs3 are not equal." << endl;
    else
        cout << "The hash_sets hs1 and hs3 are equal." << endl;
}

```

The hash_sets hs1 and hs2 are not equal.
The hash_sets hs1 and hs3 are equal.

operator==

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Tests if the hash_set object on the left side of the operator is equal to the hash_set object on the right side.

```

bool operator!=(const hash_set <Key, Traits, Allocator>& left, const hash_set <Key, Traits, Allocator>&
right);

```

Parameters

left

An object of type `hash_set`.

right

An object of type `hash_set`.

Return Value

true if the hash_set on the left side of the operator is equal to the hash_set on the right side of the operator;
otherwise **false**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. Two hash_sets are

equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hash_set_op_eq.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 == s2 )
        cout << "The hash_sets s1 and s2 are equal." << endl;
    else
        cout << "The hash_sets s1 and s2 are not equal." << endl;

    if ( s1 == s3 )
        cout << "The hash_sets s1 and s3 are equal." << endl;
    else
        cout << "The hash_sets s1 and s3 are not equal." << endl;
}
```

```
The hash_sets s1 and s2 are not equal.
The hash_sets s1 and s3 are equal.
```

operator!= (hash_multiset)

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Tests if the hash_multiset object on the left side of the operator is not equal to the hash_multiset object on the right side.

```
bool operator!=(const hash_multiset <Key, Traits, Allocator>& left, const hash_multiset <Key, Traits,
Allocator>& right);
```

Parameters

left

An object of type `hash_multiset`.

right

An object of type `hash_multiset`.

Return Value

true if the hash_multisets are not equal; **false** if hash_multisets are equal.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison between their elements. Two hash_multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hashset_op_ne.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset <int> hs1, hs2, hs3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        hs1.insert ( i );
        hs2.insert ( i * i );
        hs3.insert ( i );
    }

    if ( hs1 != hs2 )
        cout << "The hash_multisets hs1 and hs2 are not equal." << endl;
    else
        cout << "The hash_multisets hs1 and hs2 are equal." << endl;

    if ( hs1 != hs3 )
        cout << "The hash_multisets hs1 and hs3 are not equal." << endl;
    else
        cout << "The hash_multisets hs1 and hs3 are equal." << endl;
}
```

```
The hash_multisets hs1 and hs2 are not equal.
The hash_multisets hs1 and hs3 are equal.
```

operator== (hash_multiset)

NOTE

This API is obsolete. The alternative is [unordered_set Class](#).

Tests if the hash_multiset object on the left side of the operator is equal to the hash_multiset object on the right side.

```
bool operator!=(const hash_multiset <Key, Traits, Allocator>& left, const hash_multiset <Key, Traits,
Allocator>& right);
```

Parameters

left

An object of type `hash_multiset`.

right

An object of type `hash_multiset`.

Return Value

true if the `hash_multiset` on the left side of the operator is equal to the `hash_multiset` on the right side of the operator; otherwise **false**.

Remarks

The comparison between `hash_multiset` objects is based on a pairwise comparison of their elements. Two `hash_multisets` are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// hash_multiset_op_eq.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 == s2 )
        cout << "The hash_multisets s1 and s2 are equal." << endl;
    else
        cout << "The hash_multisets s1 and s2 are not equal." << endl;

    if ( s1 == s3 )
        cout << "The hash_multisets s1 and s2 are equal." << endl;
    else
        cout << "The hash_multisets s1 and s2 are not equal." << endl;
}
```

```
The hash_multisets s1 and s2 are not equal.
The hash_multisets s1 and s2 are equal.
```

See also

[`<hash_set>`](#)

<initializer_list>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class `initializer_list` and several supporting templates.

Syntax

```
#include <initializer_list>
```

Classes

CLASS	DESCRIPTION
initializer_list	Provides access to an array of elements in which each member is of the specified type.

See also

[Header Files Reference](#)

initializer_list Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Provides access to an array of elements in which each member is of the specified type.

Syntax

```
template <class Type>
class initializer_list
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The element data type to be stored in the <code>initializer_list</code> .

Remarks

An `initializer_list` can be constructed using a braced initializer list:

```
initializer_list<int> i1{ 1, 2, 3, 4 };
```

The compiler transforms braced initializer lists with homogeneous elements into an `initializer_list` whenever the function signature requires an `initializer_list`. For more details on using `initializer_list`, see [Uniform Initialization and Delegating Constructors](#)

Constructors

CONSTRUCTOR	DESCRIPTION
initializer_list	Constructs an object of type <code>initializer_list</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>value_type</code>	The type of the elements in the <code>initializer_list</code> .
<code>reference</code>	A type that provides a reference to an element in the <code>initializer_list</code> .
<code>const_reference</code>	A type that provides a constant reference to an element in the <code>initializer_list</code> .
<code>size_type</code>	A type that represents the number of elements in the <code>initializer_list</code> .
<code>iterator</code>	A type that provides an iterator for the <code>initializer_list</code> .

TYPE NAME	DESCRIPTION
const_iterator	A type that provides a constant iterator for the <code>initializer_list</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns a pointer to the first element in an <code>initializer_list</code> .
<code>end</code>	Returns a pointer to one past the last element in an <code>initializer_list</code> .
<code>size</code>	Returns the number of elements in the <code>initializer_list</code> .

Requirements

Header: `<initializer_list>`

Namespace: `std`

`initializer_list::begin`

Returns a pointer to the first element in an `initializer_list`.

```
constexpr const InputIterator* begin() const noexcept;
```

Return Value

A pointer to the first element of the `initializer_list`. If the list is empty, the pointer is the same for the beginning and end of the list.

Remarks

`initializer_list::end`

Returns a pointer to one past the last element in an `initializer list`.

```
constexpr const InputIterator* end() const noexcept;
```

Return Value

A pointer to one past the last element in the list. If the list is empty, this is the same as the pointer to the first element in the list.

`initializer_list::initializer_list`

Constructs an object of type `initializer_list`.

```
constexpr initializer_list() noexcept;
initializer_list(const InputIterator First, const InputIterator Last);
```

Parameters

PARAMETER	DESCRIPTION
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

Remarks

An `initializer_list` is based on an array of objects of the specified type. Copying an `initializer_list` creates a second instance of a list pointing to the same objects; the underlying objects are not copied.

Example

```

// initializer_list_class.cpp
// compile with: /EHsc
#include <initializer_list>
#include <iostream>

int main()
{
    using namespace std;
    // Create an empty initializer_list c0
    initializer_list<int> c0;

    // Create an initializer_list c1 with 1 element
    initializer_list<int> c1{ 3 };

    // Create an initializer_list c2 with 5 elements
    initializer_list<int> c2{ 5, 4, 3, 2, 1 };

    // Create a copy, initializer_list c3, of initializer_list c2
    initializer_list<int> c3(c2);

    // Create a initializer_list c4 by copying the range c3[ first, last)
    const int* c3_ptr = c3.begin();
    c3_ptr++;
    c3_ptr++;
    initializer_list<int> c4(c3.begin(), c3_ptr);

    // Move initializer_list c4 to initializer_list c5
    initializer_list<int> c5(move(c4));

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    cout << "c2 =";
    for (auto c : c2)
        cout << " " << c;
    cout << endl;

    cout << "c3 =";
    for (auto c : c3)
        cout << " " << c;
    cout << endl;

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;

    cout << "c5 =";
    for (auto c : c5)
        cout << " " << c;
    cout << endl;
}

```

c1 = 3 c2 = 5 4 3 2 1 c3 = 5 4 3 2 1 c4 = 5 4 c5 = 5 4

initializer_list::size

Returns the number of elements in the list.

```
constexpr size_t size() const noexcept;
```

Return Value

The number of elements in the list.

Remarks

See also

[<forward_list>](#)

<iomanip>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Include the `<iostreams>` standard header `<iomanip>` to define several manipulators that each take a single argument.

Syntax

```
#include <iomanip>
```

Remarks

Each of these manipulators returns an unspecified type, called `T1` through `T10`, that overloads both `basic_istream<Elem, Tr>::operator>>` and `basic_ostream<Elem, Tr>::operator<<`.

Manipulators

<code>get_money</code>	Obtains a monetary amount, optionally in international format.
<code>get_time</code>	Obtains a time in a time structure by using a specified format.
<code>put_money</code>	Provides a monetary amount, optionally in international format.
<code>put_time</code>	Provides a time in a time structure and a format string to use.
<code>quoted</code>	Enables convenient round-tripping of strings with insertion and extraction operators.
<code>resetiosflags</code>	Clears the specified flags.
<code>setbase</code>	Set base for integers.
<code>setfill</code>	Sets the character that will be used to fill spaces in a right-justified display.
<code>setiosflags</code>	Sets the specified flags.
<code>setprecision</code>	Sets the precision for floating-point values.
<code>setw</code>	Specifies the width of the display field.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

<iomanip> functions

11/9/2018 • 11 minutes to read • [Edit Online](#)

get_money	get_time	put_money
put_time	quoted	resetiosflags
setbase	setfill	setiosflags
setprecision	setw	

get_money

Extracts a monetary value from a stream using the desired format, and returns the value in a parameter.

```
template <class Money>
T7 get_money(Money& _Amount, bool _Intl);
```

Parameters

_Amount

The extracted monetary value.

_Intl

If **true**, use international format. The default value is **false**.

Remarks

The manipulator returns an object that, when extracted from the stream `str`, behaves as a `formatted input function` that calls the member function `get` for the locale facet `money_get` associated with `str`, using *_Intl* to indicate international format. If successful, the call stores in *_Amount* the extracted monetary value. The manipulator then returns `str`.

`Money` must be of type `long double` or an instantiation of `basic_string` with the same element and traits parameters as `str`.

get_time

Extracts a time value from a stream using a desired format. Returns the value in a parameter as a time structure.

```
template <class Elem>
T10 put_time(struct tm *_Tptr, const Elem *_Fmt);
```

Parameters

_Tptr

The time in the form of a time structure.

_Fmt

The desired format to use to obtain the time value.

Remarks

The manipulator returns an object that, when extracted from the stream `str`, behaves as a `formatted input function` that calls the member function `get` for the locale facet `time_get` associated with `str`, using `tptr` to indicate the time structure and `fmt` to indicate the beginning of a null-terminated format string. If successful, the call stores in the time structure the values associated with any extracted time fields. The manipulator then returns `str`.

put_money

Inserts a monetary amount using the desired format into a stream.

```
template <class Money>
T& put_money(const Money& _Amount, bool _Intl);
```

Parameters

_Amount

The monetary amount to insert into the stream.

_Intl

Set to **true** if manipulator should use international format, **false** if it should not.

Return Value

Returns `str`.

Remarks

The manipulator returns an object that, when inserted into the stream `str`, behaves as a formatted output function that calls the member function `put` for the locale facet `money_put` associated with `str`. If successful, the call inserts `amount` suitably formatted, using `*_Intl` to indicate international format and `str.fill()`, as the fill element. The manipulator then returns `str`.

`Money` must be of type `long double` or an instantiation of `basic_string` with the same element and traits parameters as `str`.

put_time

Writes a time value from a time structure to a stream by using a specified format.

```
template <class Elem>
T10 put_time(struct tm* _Tptr, const Elem* _Fmt);
```

Parameters

_Tptr

The time value to write to the stream, provided in a time structure.

_Fmt

The desired format to write the time value.

Remarks

The manipulator returns an object that, when inserted into the stream `str`, behaves as a `formatted output function`. The output function calls the member function `put` for the locale facet `time_put` associated with `str`. The output function uses `_Tptr` to indicate the time structure and `_Fmt` to indicate the beginning of a null-terminated format string. If successful, the call inserts literal text from the format string and converted values from the time structure. The manipulator then returns `str`.

quoted

(New in C++14) An iostream manipulator that enables convenient round-tripping of strings into and out of streams using the >> and << operators.

```
quoted(std::string str) // or wstring
quoted(const char* str) //or wchar_t*
quoted(std::string str, char delimiter, char escape) // or wide versions
quoted(const char* str, char delimiter, char escape) // or wide versions
```

Parameters

str

A std::string, char*, string literal or raw string literal, or a wide version of any of these (e.g. std::wstring, wchar_t*).

delimiter

A user-specified character, or wide character, to use as the delimiter for the beginning and end of the string.

escape

A user-specified character, or wide character, to use as the escape character for escape sequences within the string.

Remarks

See [Using Insertion Operators and Controlling Format](#).

Example

This example shows how to use `quoted` with the default delimiter and escape character using narrow strings. Wide strings are equally supported.

```

#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

void show_quoted_v_nonquoted()
{
    // Results are identical regardless of input string type:
    // string inserted { R"(This is a "sentence".)" }; // raw string literal
    // string inserted { "This is a \"sentence\"," }; // regular string literal
    const char* inserted = "This is a \"sentence\","; // const char*
    stringstream ss, ss_quoted;
    string extracted, extracted_quoted;

    ss << inserted;
    ss_quoted << quoted(inserted);

    cout << "ss.str() is storing      : " << ss.str() << endl;
    cout << "ss_quoted.str() is storing: " << ss_quoted.str() << endl << endl;

    // Round-trip the strings
    ss >> extracted;
    ss_quoted >> quoted(extracted_quoted);

    cout << "After round trip: " << endl;
    cout << "Non-quoted      : " << extracted << endl;
    cout << "Quoted          : " << extracted_quoted << endl;
}

void main(int argc, char* argv[])
{
    show_quoted_v_nonquoted();

    // Keep console window open in debug mode.
    cout << endl << "Press Enter to exit" << endl;
    string input{};
    getline(cin, input);
}

/* Output:
ss.str() is storing      : This is a "sentence".
ss_quoted.str() is storing: "This is a \"sentence\","

After round trip:
Non-quoted      : This
Quoted          : This is a "sentence".

Press Enter to exit
*/

```

Example

The following example shows how to provide custom a delimiter and/or escape character:

```

#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

void show_custom_delimiter()
{
    string inserted{ R"(This" "is" "a" "heavily-quoted" "sentence".)" };
    // string inserted{ "\"This\" \"is\" \"a\" \"heavily-quoted\" \"sentence\"" };
    // const char* inserted{ "\"This\" \"is\" \"a\" \"heavily-quoted\" \"sentence\"" };
}

```

```

stringstream ss, ss_quoted;
string extracted;

ss_quoted << quoted(inserted, '*');
ss << inserted;
cout << "ss_quoted.str() is storing: " << ss_quoted.str() << endl;
cout << "ss.str() is storing      : " << ss.str() << endl << endl;

// Use the same quoted arguments as on insertion.
ss_quoted >> quoted(extracted, '*');

cout << "After round trip: " << endl;
cout << "Quoted      : " << extracted << endl;

extracted = {};
ss >> extracted;
cout << "Non-quoted   : " << extracted << endl << endl;
}

void show_custom_escape()
{
    string inserted{ R"(\root\trunk\branch\branch\egg\yolk)" };
    // string inserted{ R"(\root\trunk\trunk\branch\branch\egg\yolk)" };
    stringstream ss, ss_quoted, ss_quoted_custom;
    string extracted;

    // Use '"' as delimiter and '~' as escape character.
    ss_quoted_custom << quoted(inserted, '"', '~');
    ss_quoted << quoted(inserted);
    ss << inserted;
    cout << "ss_quoted_custom.str(): " << ss_quoted_custom.str() << endl;
    cout << "ss_quoted.str()      : " << ss_quoted.str() << endl;
    cout << "ss.str()          : " << ss.str() << endl << endl;

    // No spaces in this string, so non-quoted behaves same as quoted
    // after round-tripping.
}

void main(int argc, char* argv[])
{
    cout << "Custom delimiter:" << endl;
    show_custom_delimiter();
    cout << "Custom escape character:" << endl;
    show_custom_escape();

    // Keep console window open in debug mode.
    cout << endl << "Press Enter to exit" << endl;
    string input{};
    getline(cin, input);
}

/* Output:
Custom delimiter:
ss_quoted.str() is storing: *This is a heavily-quoted sentence.*
ss.str() is storing      : This is a heavily-quoted sentence.

After round trip:
Quoted      : This is a heavily-quoted sentence.
Non-quoted   : This

Custom escape character:
ss_quoted_custom.str(): "\\root\\trunk\\branch\\branch\\egg\\yolk"
ss_quoted.str()      : "\\root\\trunk\\trunk\\branch\\branch\\egg\\yolk"
ss.str()          : \\root\\trunk\\branch\\branch\\egg\\yolk

Press Enter to exit
*/

```

resetiosflags

Clears the specified flags.

```
T1 resetiosflags(ios_base::fmtflags Mask);
```

Parameters

Mask

The flags to clear.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls **str**. `setf(ios_base:: fmtflags, _Mask)`, and then returns `str`.

Example

See [setw](#) for an example of using `resetiosflags`.

setbase

Set base for integers.

```
T3 setbase(int _Base);
```

Parameters

_Base

The number base.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls **str**. `setf (mask, ios_base::basefield)`, and then returns `str`. Here, `mask` is determined as follows:

- If *_Base* is 8, then `mask` is `ios_base:: oct`.
- If *_Base* is 10, then `mask` is `ios_base:: dec`.
- If *_Base* is 16, then `mask` is `ios_base:: hex`.
- If *_Base* is any other value, then `mask` is `ios_base:: fmtflags(0)`.

Example

See [setw](#) for an example of using `setbase`.

setfill

Sets the character that will be used to fill spaces in a right-justified display.

```
template <class Elem>  
T4 setfill(Elem Ch);
```

Parameters

Ch

The character that will be used to fill spaces in a right-justified display.

Return Value

The template manipulator returns an object that, when extracted from or inserted into the stream `str`, calls **str**. [fill\(ch\)](#), and then returns `str`. The type `Elem` must be the same as the element type for the stream `str`.

Example

See [setw](#) for an example of using `setfill`.

setiosflags

Sets the specified flags.

```
T2 setiosflags(ios_base::fmtflags Mask);
```

Parameters

Mask

The flags to set.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls **str**. [setf\(_Mask\)](#), and then returns `str`.

Example

See [setw](#) for an example of using `setiosflags`.

setprecision

Sets the precision for floating-point values.

```
T5 setprecision(streamsize Prec);
```

Parameters

Prec

The precision for floating-point values.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls **str**. [precision\(Prec\)](#), and then returns `str`.

Example

See [setw](#) for an example of using `setprecision`.

setw

Specifies the width of the display field for the next element in the stream.

```
T6 setw(streamsize Wide);
```

Parameters

Wide

The width of the display field.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.width(Wide)`, then returns `str`.

Remarks

`setw` sets the width only for the next element in the stream and must be inserted before each element whose width you want to specify.

Example

```
// iomanip_setw.cpp
// compile with: /EHsc
// Defines the entry point for the console application.
//
// Sample use of the following manipulators:
//  resetiosflags
//  setiosflags
//  setbase
//  setfill
//  setprecision
//  setw

#include <iostream>
#include <iomanip>

using namespace std;

const double  d1 = 1.23456789;
const double  d2 = 12.3456789;
const double  d3 = 123.456789;
const double  d4 = 1234.56789;
const double  d5 = 12345.6789;
const long    l1 = 16;
const long    l2 = 256;
const long    l3 = 1024;
const long    l4 = 4096;
const long    l5 = 65536;
int           base = 10;

void DisplayDefault( )
{
    cout << endl << "default display" << endl;
    cout << "d1 = " << d1 << endl;
    cout << "d2 = " << d2 << endl;
    cout << "d3 = " << d3 << endl;
    cout << "d4 = " << d4 << endl;
    cout << "d5 = " << d5 << endl;
}

void DisplayWidth( int n )
{
    cout << endl << "fixed width display set to " << n << ".\n";
    cout << "d1 = " << setw(n) << d1 << endl;
    cout << "d2 = " << setw(n) << d2 << endl;
    cout << "d3 = " << setw(n) << d3 << endl;
    cout << "d4 = " << setw(n) << d4 << endl;
    cout << "d5 = " << setw(n) << d5 << endl;
}

void DisplayLongs( )
{
    cout << setbase(10);
    cout << endl << "setbase(" << base << ")" << endl;
    cout << setbase(base);
    cout << "l1 = " << l1 << endl;
    cout << "l2 = " << l2 << endl;
    cout << "l3 = " << l3 << endl;
    cout << "l4 = " << l4 << endl;
```

```

    cout << 14 << endl;
    cout << "15 = " << 15 << endl;
}

int main( int argc, char* argv[] )
{
    DisplayDefault( );

    cout << endl << "setprecision(" << 3 << ")" << setprecision(3);
    DisplayDefault( );

    cout << endl << "setprecision(" << 12 << ")" << setprecision(12);
    DisplayDefault( );

    cout << setiosflags(ios_base::scientific);
    cout << endl << "setiosflags(" << ios_base::scientific << ")";
    DisplayDefault( );

    cout << resetiosflags(ios_base::scientific);
    cout << endl << "resetiosflags(" << ios_base::scientific << ")";
    DisplayDefault( );

    cout << endl << "setfill(' ' << 'S' << ' ')" << setfill('S');
    DisplayWidth(15);
    DisplayDefault( );

    cout << endl << "setfill(' ' << ' ' << ' ')" << setfill(' ');
    DisplayWidth(15);
    DisplayDefault( );

    cout << endl << "setprecision(" << 8 << ")" << setprecision(8);
    DisplayWidth(10);
    DisplayDefault( );

    base = 16;
    DisplayLongs( );

    base = 8;
    DisplayLongs( );

    base = 10;
    DisplayLongs( );

    return 0;
}

```

```

default display
d1 = 1.23457
d2 = 12.3457
d3 = 123.457
d4 = 1234.57
d5 = 12345.7

setprecision(3)
default display
d1 = 1.23
d2 = 12.3
d3 = 123
d4 = 1.23e+003
d5 = 1.23e+004

setprecision(12)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789

```

```
d5 = 12345.6789
```

```
setiosflags(4096)
default display
d1 = 1.234567890000e+000
d2 = 1.234567890000e+001
d3 = 1.234567890000e+002
d4 = 1.234567890000e+003
d5 = 1.234567890000e+004
```

```
resetiosflags(4096)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789
```

```
setfill('S')
fixed width display set to 15.
d1 = SSSSS1.23456789
d2 = SSSSS12.3456789
d3 = SSSSS123.456789
d4 = SSSSS1234.56789
d5 = SSSSS12345.6789
```

```
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789
```

```
setfill(' ')
fixed width display set to 15.
d1 =      1.23456789
d2 =      12.3456789
d3 =      123.456789
d4 =      1234.56789
d5 =      12345.6789
```

```
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789
```

```
setprecision(8)
fixed width display set to 10.
d1 = 1.2345679
d2 = 12.345679
d3 = 123.45679
d4 = 1234.5679
d5 = 12345.679
```

```
default display
d1 = 1.2345679
d2 = 12.345679
d3 = 123.45679
d4 = 1234.5679
d5 = 12345.679
```

```
setbase(16)
11 = 10
12 = 100
13 = 400
14 = 1000
15 = 10000
```



```
15 = 10000
```

```
setbase(8)
```

```
11 = 20
```

```
12 = 400
```

```
13 = 2000
```

```
14 = 10000
```

```
15 = 200000
```

```
setbase(10)
```

```
11 = 16
```

```
12 = 256
```

```
13 = 1024
```

```
14 = 4096
```

```
15 = 65536
```

See also

[<iomanip>](#)

<ios>

11/9/2018 • 3 minutes to read • [Edit Online](#)

Defines several types and functions basic to the operation of iostreams. This header is typically included for you by another iostream headers; you rarely include it directly.

Syntax

```
#include <ios>
```

Remarks

A large group of functions are manipulators. A manipulator declared in <ios> alters the values stored in its argument object of class [ios_base](#). Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the template classes [basic_istream](#) or [basic_ostream](#). For example, [noskipws\(str\)](#) clears the format flag `ios_base::skipws` in the object `str`, which can be of one of these types.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, because of special insertion and extraction operations supplied for the classes derived from `ios_base`. For example:

```
istr>> noskipws;
```

calls [noskipws\(istr\)](#).

Typedefs

TYPE NAME	DESCRIPTION
ios	Supports the ios class from the old iostream library.
streamoff	Supports internal operations.
streampos	Holds the current position of the buffer pointer or file pointer.
streamsize	Specifies the size of the stream.
wios	Supports the wios class from the old iostream library.
wstreampos	Holds the current position of the buffer pointer or file pointer.

Manipulators

boolalpha	Specifies that variables of type bool appear as true or false in the stream.
dec	Specifies that integer variables appear in base 10 notation.

defaultfloat	Configures the flags of an <code>ios_base</code> object to use a default display format for float values.
fixed	Specifies that a floating-point number is displayed in fixed-decimal notation.
hex	Specifies that integer variables appear in base 16 notation.
internal	Causes a number's sign to be left justified and the number to be right justified.
left	Causes text that is not as wide as the output width to appear in the stream flush with the left margin.
noboolalpha	Specifies that variables of type <code>bool</code> appear as 1 or 0 in the stream.
noshowbase	Turns off indicating the notational base in which a number is displayed.
noshowpoint	Displays only the whole-number part of floating-point numbers whose fractional part is zero.
noshowpos	Causes positive numbers to not be explicitly signed.
noskipws	Cause spaces to be read by the input stream.
nounitbuf	Causes output to be buffered and processed when the buffer is full.
nouppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.
oct	Specifies that integer variables appear in base 8 notation.
right	Causes text that is not as wide as the output width to appear in the stream flush with the right margin.
scientific	Causes floating point numbers to be displayed using scientific notation.
showbase	Indicates the notational base in which a number is displayed.
showpoint	Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.
showpos	Causes positive numbers to be explicitly signed.
skipws	Cause spaces to not be read by the input stream.
unitbuf	Causes output to be processed when the buffer is not empty.

uppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

Classes

CLASS	DESCRIPTION
basic_ios	The template class describes the storage and member functions common to both input streams (of template class basic_istream) and output streams (of template class basic_ostream) that depend on the template parameters.
fpos	The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream.
ios_base	The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<ios> functions

11/9/2018 • 12 minutes to read • [Edit Online](#)

defaultfloat	boolalpha	dec
fixed	hex	internal
left	noboolalpha	noshowbase
noshowpoint	noshowpos	noskipws
nounitbuf	nouppercase	oct
right	scientific	showbase
showpoint	showpos	skipws
unitbuf	uppercase	

boolalpha

Specifies that variables of type `bool` appear as **true** or **false** in the stream.

```
ios_base& boolalpha(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, variables of type **bool** are displayed as 1 or 0.

`boolalpha` effectively calls `str.setf(ios_base::boolalpha)`, and then returns *str*.

`noboolalpha` reverses the effect of `boolalpha`.

Example

```
// ios_boolalpha.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    bool b = true;
    cout << b << endl;
    boolalpha( cout );
    cout << b << endl;
    noboolalpha( cout );
    cout << b << endl;
    cout << boolalpha << b << endl;
}
```

```
1
true
1
true
```

dec

Specifies that integer variables appear in base 10 notation.

```
ios_base& dec(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, integer variables are displayed in base 10.

`dec` effectively calls `str.setf(ios_base::dec , ios_base::basefield)`, and then returns *str*.

Example

```
// ios_dec.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int i = 100;

    cout << i << endl;    // Default is base 10
    cout << hex << i << endl;
    dec( cout );
    cout << i << endl;
    oct( cout );
    cout << i << endl;
    cout << dec << i << endl;
}
```

```
100
64
100
144
100
```

<ios> defaultfloat

Configures the flags of an `ios_base` object to use a default display format for float values.

```
ios_base& defaultfloat(ios_base& _Iosbase);
```

Parameters

_Iosbase

An `ios_base` object.

Remarks

The manipulator effectively calls `_Iosbase.ios_base::unsetf(ios_base::floatfield)`, then returns `_Iosbase`.

fixed

Specifies that a floating-point number is displayed in fixed-decimal notation.

```
ios_base& fixed(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

`fixed` is the default display notation for floating-point numbers. `scientific` causes floating-point numbers to be displayed using scientific notation.

The manipulator effectively calls `str.setf(ios_base::fixed, ios_base::floatfield)`, and then returns `str`.

Example

```
// ios_fixed.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    float i = 1.1F;

    cout << i << endl;    // fixed is the default
    cout << scientific << i << endl;
    cout.precision( 1 );
    cout << fixed << i << endl;
}
```

```
1.1
1.100000e+000
1.1
```

hex

Specifies that integer variables shall appear in base 16 notation.

```
ios_base& hex(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, integer variables are displayed in base 10 notation. [dec](#) and [oct](#) also change the way integer variables appear.

The manipulator effectively calls `str.setf(ios_base::hex, ios_base::basefield)`, and then returns *str*.

Example

See [dec](#) for an example of how to use `hex`.

internal

Causes a number's sign to be left justified and the number to be right justified.

```
ios_base& internal(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

[showpos](#) causes the sign to display for positive numbers.

The manipulator effectively calls `str.setf(ios_base::internal, ios_base::adjustfield)`, and then returns *str*.

Example


```
// ios_internal.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>

int main( void )
{
    using namespace std;
    float i = -123.456F;
    cout.fill( '.' );
    cout << setw( 10 ) << i << endl;
    cout << setw( 10 ) << internal << i << endl;
}
```

```
..-123.456
-..123.456
```

left

Causes text that is not as wide as the output width to appear in the stream flush with the left margin.

```
ios_base& left(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

The manipulator effectively calls `str.setf(ios_base::left, ios_base::adjustfield)`, and then returns *str*.

Example

```
// ios_left.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    double f1= 5.00;
    cout.width( 20 );
    cout << f1 << endl;
    cout << left << f1 << endl;
}
```

```
5
5
```

noboolalpha

Specifies that variables of type `bool` appear as 1 or 0 in the stream.

```
ios_base& noboolalpha(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, `noboolalpha` is in effect.

`noboolalpha` effectively calls `str.unsetf(ios_base::boolalpha)`, and then returns *str*.

[boolalpha](#) reverses the effect of `noboolalpha`.

Example

See [boolalpha](#) for an example of using `noboolalpha`.

noshowbase

Turns off indicating the notational base in which a number is displayed.

```
ios_base& noshowbase(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

`noshowbase` is on by default. Use [showbase](#) to indicate the notational base of numbers.

The manipulator effectively calls `str.unsetf(ios_base::showbase)`, and then returns *str*.

Example

See [showbase](#) for an example of how to use `noshowbase`.

noshowpoint

Displays only the whole-number part of floating-point numbers whose fractional part is zero.

```
ios_base& noshowpoint(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

`noshowpoint` is on by default; use [showpoint](#) and [precision](#) to display zeros after the decimal point.

The manipulator effectively calls `str.unsetf(ios_base::showpoint)`, and then returns *str*.

Example

```
// ios_noshowpoint.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    double f1= 5.000;
    cout << f1 << endl;    // noshowpoint is default
    cout.precision( 4 );
    cout << showpoint << f1 << endl;
    cout << noshowpoint << f1 << endl;
}
```

```
5
5.000
5
```

noshowpos

Causes positive numbers to not be explicitly signed.

```
ios_base& noshowpos(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

`noshowpos` is on by default.

The manipulator effectively calls `str.unsetf(ios_base::showps)`, then returns *str*.

Example

See [showpos](#) for an example of using `noshowpos`.

noskipws

Cause spaces to be read by the input stream.

```
ios_base& noskipws(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, `skipws` is in effect. When a space is read in the input stream, it signals the end of the buffer.

The manipulator effectively calls `str.unsetf(ios_base::skipws)`, and then returns `str`.

Example

```
// ios_noskipws.cpp
// compile with: /EHsc
#include <iostream>
#include <string>

int main() {
    using namespace std;
    string s1, s2, s3;
    cout << "Enter three strings: ";
    cin >> noskipws >> s1 >> s2 >> s3;
    cout << "." << s1 << "." << endl;
    cout << "." << s2 << "." << endl;
    cout << "." << s3 << "." << endl;
}
```

nounitbuf

Causes output to be buffered and processed on when the buffer is full.

```
ios_base& nounitbuf(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

`unitbuf` causes the buffer to be processed when it is not empty.

The manipulator effectively calls `str.unsetf(ios_base::unitbuf)`, and then returns `str`.

nouppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.

```
ios_base& nouppercase(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_ Str` is derived.

Remarks

The manipulator effectively calls `str.unsetf(ios_base::uppercase)`, and then returns *str*.

Example

See [uppercase](#) for an example of using `nouppercase`.

oct

Specifies that integer variables appear in base 8 notation.

```
ios_base& oct(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, integer variables are displayed in base 10 notation. [dec](#) and [hex](#) also change the way integer variables appear.

The manipulator effectively calls `str.setf(ios_base::oct , ios_base::basefield)`, and then returns *str*.

Example

See [dec](#) for an example of how to use `oct`.

right

Causes text that is not as wide as the output width to appear in the stream flush with the right margin.

```
ios_base& right(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

[left](#) also modifies the justification of text.

The manipulator effectively calls `str.setf(ios_base::right , ios_base::adjustfield)`, and then returns *str*.

Example

```
// ios_right.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    double f1= 5.00;
    cout << f1 << endl;
    cout.width( 20 );
    cout << f1 << endl;
    cout.width( 20 );
    cout << left << f1 << endl;
    cout.width( 20 );
    cout << f1 << endl;
    cout.width( 20 );
    cout << right << f1 << endl;
    cout.width( 20 );
    cout << f1 << endl;
}
```

```
5
           5
5
5
           5
           5
```

scientific

Causes floating-point numbers to be displayed using scientific notation.

```
ios_base& scientific(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, `fixed` notation is in effect for floating-point numbers.

The manipulator effectively calls `str.setf(ios_base::scientific, ios_base::floatfield)`, and then returns *str*.

Example

```
// ios_scientific.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    float i = 100.23F;

    cout << i << endl;
    cout << scientific << i << endl;
}
```

```
100.23
1.002300e+002
```

showbase

Indicates the notational base in which a number is displayed.

```
ios_base& showbase(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

The notational base of a number can be changed with [dec](#), [oct](#), or [hex](#).

The manipulator effectively calls `str.setf(ios_base::showbase)`, and then returns *str*.

Example

```
// ios_showbase.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int j = 100;

    cout << showbase << j << endl;    // dec is default
    cout << hex << j << showbase << endl;
    cout << oct << j << showbase << endl;

    cout << dec << j << noshowbase << endl;
    cout << hex << j << noshowbase << endl;
    cout << oct << j << noshowbase << endl;
}
```

```
100
0x64
0144
100
64
144
```

showpoint

Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.

```
ios_base& showpoint(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

By default, [noshowpoint](#) is in effect.

The manipulator effectively calls `str.setf(ios_base::showpoint)`, and then returns *str*.

Example

See [noshowpoint](#) for an example of using `showpoint`.

showpos

Causes positive numbers to be explicitly signed.

```
ios_base& showpos(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which `_Str` is derived.

Remarks

[noshowpos](#) is the default.

The manipulator effectively calls `str.setf(ios_base::showpos)`, and then returns *str*.

Example


```
// ios_showpos.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int i = 1;

    cout << noshowpos << i << endl;    // noshowpos is default
    cout << showpos << i << endl;
}
```

```
1
+1
```

skipws

Cause spaces to not be read by the input stream.

```
ios_base& skipws(ios_base& str);
```

Parameters

str

A reference to an object of type [ios_base](#), or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *_Str* is derived.

Remarks

By default, `skipws` is in effect. [noskipws](#) will cause spaces to be read from the input stream.

The manipulator effectively calls `str.setf(ios_base::skipws)`, and then returns *str*.

Example

```
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    char s1, s2, s3;
    cout << "Enter three characters: ";
    cin >> skipws >> s1 >> s2 >> s3;
    cout << "." << s1 << "." << endl;
    cout << "." << s2 << "." << endl;
    cout << "." << s3 << "." << endl;
}
```

```
1 2 3
```

```
Enter three characters: 1 2 3
```

```
.1.  
.2.  
.3.
```

unitbuf

Causes output to be processed when the buffer is not empty.

```
ios_base& unitbuf(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

Note that `endl` also flushes the buffer.

`nounitbuf` is in effect by default.

The manipulator effectively calls `str.setf(ios_base::unitbuf)`, and then returns *str*.

uppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

```
ios_base& uppercase(ios_base& str);
```

Parameters

str

A reference to an object of type `ios_base`, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, `nouppercase` is in effect.

The manipulator effectively calls `str.setf(ios_base::uppercase)`, and then returns *str*.

Example

```
// ios_uppercase.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
    using namespace std;

    double i = 1.23e100;
    cout << i << endl;
    cout << uppercase << i << endl;

    int j = 10;
    cout << hex << nouppercase << j << endl;
    cout << hex << uppercase << j << endl;
}
```

```
1.23e+100
1.23E+100
a
A
```

See also

[<ios>](#)

<ios> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

ios	streamoff	streampos
streamsize	wios	wstreampos

ios

Supports the ios class from the old iostream library.

```
typedef basic_ios<char, char_traits<char>> ios;
```

Remarks

The type is a synonym for template class [basic_ios](#), specialized for elements of type **char** with default character traits.

streamoff

Supports internal operations.

```
#ifdef _WIN64
    typedef __int64 streamoff;
#else
    typedef long streamoff;
#endif
```

Remarks

The type is a signed integer that describes an object that can store a byte offset involved in various stream positioning operations. Its representation has at least 32 value bits. It is not necessarily large enough to represent an arbitrary byte position within a stream. The value `streamoff(-1)` generally indicates an erroneous offset.

streampos

Holds the current position of the buffer pointer or file pointer.

```
typedef fpos<mbstate_t> streampos;
```

Remarks

The type is a synonym for `fpos<mbstate_t>`.

Example

```
// ios_streampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;

    ofstream x( "iostream.txt" );
    x << "testing";
    streampos y = x.tellp( );
    cout << y << endl;
}
```

7

streamsize

Denotes the size of the stream.

```
#ifndef _WIN64
    typedef __int64 streamsize;
#else
    typedef int streamsize;
#endif
```

Remarks

The type is a signed integer that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It is not necessarily large enough to represent an arbitrary byte position within a stream.

Example

After compiling and running the following program, look at the file test.txt to see the effect of setting `streamsize`.

```
// ios_streamsize.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    char a[16] = "any such text";
    ofstream x( "test.txt" );
    streamsize y = 6;
    x.write( a, y );
}
```

wios

Supports the wios class from the old iostream library.

```
typedef basic_ios<wchar_t, char_traits<wchar_t>> wios;
```

Remarks

The type is a synonym for template class [basic_ios](#), specialized for elements of type **wchar_t** with default character traits.

wstreampos

Holds the current position of the buffer pointer or file pointer.

```
typedef fpos<mbstate_t> wstreampos;
```

Remarks

The type is a synonym for [fpos](#)< `mbstate_t` >.

Example

```
// ios_wstreampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    wofstream xw( "wiostream.txt" );
    xw << L"testing";
    wstreampos y = xw.tellp( );
    cout << y << endl;
}
```

7

See also

[<ios>](#)

basic_ios Class

11/9/2018 • 12 minutes to read • [Edit Online](#)

The template class describes the storage and member functions common to both input streams (of template class [basic_istream](#)) and output streams (of template class [basic_ostream](#)) that depend on the template parameters. (The class [ios_base](#) describes what is common and not dependent on template parameters.) An object of class **basic_ios<class Elem, class Traits>** helps control a stream with elements of type `Elem`, whose character traits are determined by the class `Traits`.

Syntax

```
template <class Elem, class Traits>
class basic_ios : public ios_base
```

Parameters

Elem

A type.

Traits

A variable of type `char_traits`.

Remarks

An object of class **basic_ios<class Elem, class Traits>** stores:

- A tie pointer to an object of type [basic_istream<Elem, Traits>](#).
- A stream buffer pointer to an object of type [basic_streambuf<Elem, Traits >](#).
- [Formatting information](#).
- [Stream state information](#) in a base object of type [ios_base](#).
- A fill character in an object of type `char_type`.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_ios	Constructs the <code>basic_ios</code> class.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A synonym for the template parameter <code>Elem</code> .
int_type	A synonym for <code>Traits::int_type</code> .
off_type	A synonym for <code>Traits::off_type</code> .

TYPE NAME	DESCRIPTION
<code>pos_type</code>	A synonym for <code>Traits::pos_type</code> .
<code>traits_type</code>	A synonym for the template parameter <code>Traits</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>bad</code>	Indicates a loss of integrity of the stream buffer.
<code>clear</code>	Clears all error flags.
<code>copyfmt</code>	Copies flags from one stream to another.
<code>eof</code>	Indicates if the end of a stream has been reached.
<code>exceptions</code>	Indicates which exceptions will be thrown by the stream.
<code>fail</code>	Indicates failure to extract a valid field from a stream.
<code>fill</code>	Specifies or returns the character that will be used when the text is not as wide as the stream.
<code>good</code>	Indicates the stream is in good condition.
<code>imbue</code>	Changes the locale.
<code>init</code>	Called by <code>basic_ios</code> constructors.
<code>move</code>	Moves all values, except the pointer to the stream buffer, from the parameter to the current object.
<code>narrow</code>	Finds the equivalent char to a given <code>char_type</code> .
<code>rdbuf</code>	Routes stream to specified buffer.
<code>rdstate</code>	Reads the state of bits for flags.
<code>set_rdbuf</code>	Assigns a stream buffer to be the read buffer for this stream object.
<code>setstate</code>	Sets additional flags.
<code>swap</code>	Exchanges the values in this <code>basic_ios</code> object for those of another <code>basic_ios</code> object. The pointers to the stream buffers are not swapped.
<code>tie</code>	Ensures that one stream is processed before another stream.
<code>widen</code>	Finds the equivalent <code>char_type</code> to a given char.

Operators

OPERATOR	DESCRIPTION
explicit operator bool	Allows use of a <code>basic_ios</code> object as a bool . Automatic type conversion is disabled to prevent common, unintended side effects.
operator void *	Indicates if the stream is still good.
operator!	Indicates if the stream is not bad.

Requirements

Header: `<ios>`

Namespace: `std`

`basic_ios::bad`

Indicates a loss of integrity of the stream buffer

```
bool bad() const;
```

Return Value

true if `rdstate & badbit` is nonzero; otherwise **false**.

For more information on `badbit`, see [ios_base::iostate](#).

Example

```
// basic_ios_bad.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
    using namespace std;
    bool b = cout.bad( );
    cout << boolalpha;
    cout << b << endl;

    b = cout.good( );
    cout << b << endl;
}
```

`basic_ios::basic_ios`

Constructs the `basic_ios` class.

```
explicit basic_ios(basic_streambuf<Elem, Traits>* sb);
basic_ios();
```

Parameters

sb

Standard buffer to store input or output elements.

Remarks

The first constructor initializes its member objects by calling `init(_ Sb)`. The second (protected) constructor leaves its member objects uninitialized. A later call to `init` must initialize the object before it can be safely destroyed.

basic_ios::char_type

A synonym for the template parameter `Elem`.

```
typedef Elem char_type;
```

basic_ios::clear

Clears all error flags.

```
void clear(iostate state = goodbit, bool reraise = false);  
void clear(io_state state);
```

Parameters

state

(Optional) The flags you want to set after clearing all flags. Defaults to `goodbit`.

reraise

(Optional) Specifies whether the exception should be re-raised. Defaults to **false** (will not re-raise the exception).

Remarks

The flags are `goodbit`, `failbit`, `eofbit`, and `badbit`. Test for these flags with `good`, `bad`, `eof`, and `fail`.

The member function replaces the stored stream state information with:

```
state | ( ( rdbuf != 0 ? goodbit : badbit )
```

If `state` & `exceptions` is nonzero, it then throws an object of class `failure`.

Example

See `rdstate` and `getline` for examples using `clear`.

basic_ios::copyfmt

Copies flags from one stream to another.

```
basic_ios<Elem, Traits>& copyfmt(  
    const basic_ios<Elem, Traits>& right);
```

Parameters

right

The stream whose flags you want to copy.

Return Value

The **this** object for the stream to which you are copying the flags.

Remarks

The member function reports the callback event **erase_event**. It then copies from *right* into ***this** the fill character, the tie pointer, and the formatting information. Before altering the exception mask, it reports the callback event `copyfmt_event`. If, after the copy is complete, **state & exceptions** is nonzero, the function effectively calls `clear` with the argument `rdstate`. It returns ***this**.

Example

```
// basic_ios_copyfmt.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    ofstream x( "test.txt" );
    int i = 10;

    x << showpos;
    cout << i << endl;
    cout.copyfmt( x );
    cout << i << endl;
}
```

basic_ios::eof

Indicates if the end of a stream has been reached.

```
bool eof() const;
```

Return Value

true if the end of the stream has been reached, **false** otherwise.

Remarks

The member function returns **true** if `rdstate` & `eofbit` is nonzero. For more information on `eofbit`, see [ios_base::iostate](#).

Example

```
// basic_ios_eof.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char* argv[] )
{
    fstream fs;
    int n = 1;
    fs.open( "basic_ios_eof.txt" ); // an empty file
    cout << boolalpha
    cout << fs.eof() << endl;
    fs >> n; // Read the char in the file
    cout << fs.eof() << endl;
}
```

basic_ios::exceptions

Indicates which exceptions will be thrown by the stream.

```
iostate exceptions() const;
void exceptions(iostate Newexcept);
void exceptions(io_state Newexcept);
```

Parameters

Newexcept

The flags that you want to throw an exception.

Return Value

The flags that are currently specified to throw an exception for the stream.

Remarks

The first member function returns the stored exception mask. The second member function stores *_Except* in the exception mask and returns its previous stored value. Note that storing a new exception mask can throw an exception just like the call [clear\(rdstate \)](#).

Example

```
// basic_ios_exceptions.cpp
// compile with: /EHsc /GR
#include <iostream>

int main( )
{
    using namespace std;

    cout << cout.exceptions( ) << endl;
    cout.exceptions( ios::eofbit );
    cout << cout.exceptions( ) << endl;
    try
    {
        cout.clear( ios::eofbit );    // Force eofbit on
    }
    catch ( exception &e )
    {
        cout.clear( );
        cout << "Caught the exception." << endl;
        cout << "Exception class: " << typeid(e).name() << endl;
        cout << "Exception description: " << e.what() << endl;
    }
}
```

```
0
1
Caught the exception.
Exception class: class std::ios_base::failure
Exception description: ios_base::eofbit set
```

basic_ios::fail

Indicates failure to extract a valid field from a stream.

```
bool fail() const;
```

Return Value

true if `rdstate` & (`badbit`|`failbit`) is nonzero, otherwise **false**.

For more information on `failbit`, see [ios_base::iostate](#).

Example

```
// basic_ios_fail.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
    using namespace std;
    bool b = cout.fail( );
    cout << boolalpha;
    cout << b << endl;
}
```

basic_ios::fill

Specifies or returns the character that will be used when the text is not as wide as the stream.

```
char_type fill() const;
char_type fill(char_type Char);
```

Parameters

Char

The character you want as the fill character.

Return Value

The current fill character.

Remarks

The first member function returns the stored fill character. The second member function stores *Char* in the fill character and returns its previous stored value.

Example

```
// basic_ios_fill.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;

    cout << setw( 5 ) << 'a' << endl;
    cout.fill( 'x' );
    cout << setw( 5 ) << 'a' << endl;
    cout << cout.fill( ) << endl;
}
```

```
a
xxxxa
x
```

basic_ios::good

Indicates the stream is in good condition.

```
bool good() const;
```

Return Value

true if `rdstate` `==` `goodbit` (no state flags are set), otherwise, **false**.

For more information on `goodbit`, see [ios_base::iostate](#).

Example

See [basic_ios::bad](#) for an example of using `good`.

basic_ios::imbue

Changes the locale.

```
locale imbue(const locale& Loc);
```

Parameters

Loc

A locale string.

Return Value

The previous locale.

Remarks

If `rdbuf` is not a null pointer, the member function calls

```
rdbuf -> pubimbue(_ Loc)
```

In any case, it returns [ios_base::imbue\(_ Loc\)](#).

Example

```
// basic_ios_imbue.cpp
// compile with: /EHsc
#include <iostream>
#include <locale>

int main( )
{
    using namespace std;

    cout.imbue( locale( "french_france" ) );
    double x = 1234567.123456;
    cout << x << endl;
}
```

basic_ios::init

Called by `basic_ios` constructors.

```
void init(basic_streambuf<Elem,Traits>* _Sb, bool _Isstd = false);
```

Parameters

_Sb

Standard buffer to store input or output elements.

_Isstd

Specifies whether this is a standard stream.

Remarks

The member function stores values in all member objects, so that:

- `rdbuf` returns *_Sb*.
- `tie` returns a null pointer.
- `rdstate` returns `goodbit` if *_Sb* is nonzero; otherwise, it returns `badbit`.
- `exceptions` returns `goodbit`.
- `flags` returns `skipws | dec`.
- `width` returns 0.
- `precision` returns 6.
- `fill` returns the space character.
- `getloc` returns `locale::classic`.
- `isword` returns zero, and `pword` returns a null pointer for all argument values.

`basic_ios::int_type`

A synonym for `traits_type::int_type`.

```
typedef typename traits_type::int_type int_type;
```

`basic_ios::move`

Moves all values, except the pointer to the stream buffer, from the parameter to the current object.

```
void move(basic_ios&& right);
```

Parameters

right

The `ios_base` object to move values from.

Remarks

The protected member function moves all the values stored in *right* to `*this` except the stored `stream buffer pointer`, which is unchanged in *right* and set to a null pointer in `*this`. The stored `tie pointer` is set to a null pointer in *right*.

`basic_ios::narrow`

Finds the equivalent char to a given `char_type`.

```
char narrow(char_type Char, char Default = '\0') const;
```

Parameters

Char

The **char** to convert.

Default

The **char** that you want returned if no equivalent is found.

Return Value

The equivalent **char** to a given `char_type`.

Remarks

The member function returns `use_facet<ctype<E>>(getloc()).narrow(Char, Default)`.

Example

```
// basic_ios_narrow.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
    using namespace std;
    wchar_t *x = L"test";
    char y[10];
    cout << x[0] << endl;
    wcout << x << endl;
    y[0] = wcout.narrow( x[0] );
    cout << y[0] << endl;
}
```

basic_ios::off_type

A synonym for `traits_type::off_type`.

```
typedef typename traits_type::off_type off_type;
```

basic_ios::operator void *

Indicates if the stream is still good.

```
operator void *() const;
```

Return Value

The operator returns a null pointer only if [fail](#).

Example


```
// basic_ios_opgood.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << (bool)(&cout != 0) << endl;    // Stream is still good
}
```

1

basic_ios::operator!

Indicates if the stream is not bad.

```
bool operator!() const;
```

Return Value

Returns [fail](#).

Example

```
// basic_ios_opbad.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << !cout << endl;    // Stream is not bad
}
```

0

basic_ios::operator bool

Allows use of a `basic_ios` object as a **bool**. Automatic type conversion is disabled to prevent common, unintended side effects.

```
explicit operator bool() const;
```

Remarks

The operator returns a value convertible to **false** only if `fail()`. The return type is convertible only to **bool**, not to `void *` or other known scalar type.

basic_ios::pos_type

A synonym for `traits_type::pos_type`.

```
typedef typename traits_type::pos_type pos_type;
```

basic_ios::rdbuf

Routes stream to specified buffer.

```
basic_streambuf<Elem, Traits> *rdbuf() const;  
basic_streambuf<Elem, Traits> *rdbuf(  
    basic_streambuf<Elem, Traits>* _Sb);
```

Parameters

_Sb

A stream.

Remarks

The first member function returns the stored stream buffer pointer.

The second member function stores *_Sb* in the stored stream buffer pointer and returns the previously stored value.

Example

```
// basic_ios_rdbuf.cpp  
// compile with: /EHsc  
#include <ios>  
#include <iostream>  
#include <fstream>  
  
int main( )  
{  
    using namespace std;  
    ofstream file( "rdbuf.txt" );  
    streambuf *x = cout.rdbuf( file.rdbuf( ) );  
    cout << "test" << endl;    // Goes to file  
    cout.rdbuf(x);  
    cout << "test2" << endl;  
}
```

```
test2
```

basic_ios::rdstate

Reads the state of bits for flags.

```
iosstate rdstate() const;
```

Return Value

The stored stream state information.

Example

```
// basic_ios_rdtype.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>
using namespace std;

void TestFlags( ios& x )
{
    cout << ( x.rdstate( ) & ios::badbit ) << endl;
    cout << ( x.rdstate( ) & ios::failbit ) << endl;
    cout << ( x.rdstate( ) & ios::eofbit ) << endl;
    cout << endl;
}

int main( )
{
    fstream x( "c:\\test.txt", ios::out );
    x.clear( );
    TestFlags( x );
    x.clear( ios::badbit | ios::failbit | ios::eofbit );
    TestFlags( x );
}
```

```
0
0
0

4
2
1
```

basic_ios::setstate

Sets additional flags.

```
void setstate(iostate _State);
```

Parameters

_State

Additional flags to set.

Remarks

The member function effectively calls [clear](#)(*_State* | [rdstate](#)).

Example

```
// basic_ios_setstate.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
using namespace std;

int main( )
{
    bool b = cout.bad( );
    cout << b << endl;    // Good
    cout.clear( ios::badbit );
    b = cout.bad( );
    // cout.clear( );
    cout << b << endl;    // Is bad, good
    b = cout.fail( );
    cout << b << endl;    // Not failed
    cout.setstate( ios::failbit );
    b = cout.fail( );
    cout.clear( );
    cout << b << endl;    // Is failed, good
    return 0;
}
```

```
0
1
```

basic_ios::set_rdbuf

Assigns a stream buffer to be the read buffer for this stream object.

```
void set_rdbuf(
    basic_streambuf<Elem, Tr>* strbuf)
```

Parameters

strbuf

The stream buffer to become the read buffer.

Remarks

The protected member function stores *strbuf* in the `stream buffer pointer`. It does not call `clear`.

basic_ios::tie

Ensures that one stream is processed before another stream.

```
basic_ostream<Elem, Traits> *tie() const;
basic_ostream<Elem, Traits> *tie(
    basic_ostream<Elem, Traits>* str);
```

Parameters

str

A stream.

Return Value

The first member function returns the stored tie pointer. The second member function stores *str* in the tie pointer and returns its previous stored value.

Remarks

`tie` causes two streams to be synchronized, such that, operations on one stream occur after operations on the other stream are complete.

Example

In this example, by tying `cin` to `cout`, it is guaranteed that the "Enter a number:" string will go to the console before the number itself is extracted from `cin`. This eliminates the possibility that the "Enter a number:" string is still sitting in the buffer when the number is read, so that we are certain that the user actually has some prompt to respond to. By default, `cin` and `cout` are tied.

```
#include <ios>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    cin.tie( &cout );
    cout << "Enter a number:";
    cin >> i;
}
```

basic_ios::traits_type

A synonym for the template parameter `Traits`.

```
typedef Traits traits_type;
```

basic_ios::widen

Finds the equivalent `char_type` to a given **char**.

```
char_type widen(char Char) const;
```

Parameters

Char

The character to convert.

Return Value

Finds the equivalent `char_type` to a given **char**.

Remarks

The member function returns `use_facet< ctype< E> > (getloc). widen (Char)`.

Example

```
// basic_ios_widen.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
    using namespace std;
    char *z = "Hello";
    wchar_t y[2] = {0,0};
    cout << z[0] << endl;
    y[0] = wcout.widen( z[0] );
    wcout << &y[0] << endl;
}
```

basic_ios::swap

Exchanges the values in this `basic_ios` object for those of another `basic_ios` object. However, the pointers to the stream buffers are not swapped.

```
void swap(basic_ios&& right);
```

Parameters

right

The `basic_ios` object that is used to exchange values.

Remarks

The protected member function exchanges all the values stored in *right* with `*this` except the stored stream buffer pointer.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

fpos Class

3/28/2019 • 4 minutes to read • [Edit Online](#)

The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream. An object of class `fpos< St>` effectively stores at least two member objects:

- A byte offset, of type `streamoff`.
- A conversion state, for use by an object of class `basic_filebuf`, of type `St`, typically `mbstate_t`.

It can also store an arbitrary file position, for use by an object of class `basic_filebuf`, of type `fpos_t`. For an environment with limited file size, however, `streamoff` and `fpos_t` may sometimes be used interchangeably. For an environment with no streams that have a state-dependent encoding, `mbstate_t` may actually be unused. Therefore, the number of member objects stored may vary.

Syntax

```
template <class Statetype>
class fpos
```

Parameters

Statetype

State information.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>fpos</code>	Create an object that contains information about a position (offset) in a stream.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>seekpos</code>	Used internally by the C++ Standard Library only. Do not call this method from your code.
<code>state</code>	Sets or returns the conversion state.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests file-position indicators for inequality.
<code>operator+</code>	Increments a file-position indicator.
<code>operator+=</code>	Increments a file-position indicator.

OPERATOR	DESCRIPTION
<code>operator-</code>	Decrements a file-position indicator.
<code>operator-=</code>	Decrements a file-position indicator.
<code>operator==</code>	Tests file-position indicators for equality.
<code>operator streamoff</code>	Casts object of type <code>fpos</code> to object of type <code>streamoff</code> .

Requirements

Header: `<ios>`

Namespace: `std`

`fpos::fpos`

Create an object that contains information about a position (offset) in a stream.

```
fpos(streamoff _Off = 0);

fpos(Statetype _State, fpos_t _Filepos);
```

Parameters

_Off

The offset into the stream.

_State

The starting state of the `fpos` object.

_Filepos

The offset into the stream.

Remarks

The first constructor stores the offset *_Off*, relative to the beginning of file and in the initial conversion state (if that matters). If *_Off* is -1, the resulting object represents an invalid stream position.

The second constructor stores a zero offset and the object *_State*.

`fpos::operator!=`

Tests file-position indicators for inequality.

```
bool operator!=(const fpos<Statetype>& right) const;
```

Parameters

right

The file-position indicator against which to compare.

Return Value

true if the file-position indicators are not equal, otherwise **false**.

Remarks

The member function returns `!(*this == right)`.

Example

```
// fpos_op_neq.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
    using namespace std;

    fpos<int> pos1, pos2;
    ifstream file;
    char c;

    // Compare two fpos object
    if ( pos1 != pos2 )
        cout << "File position pos1 and pos2 are not equal" << endl;
    else
        cout << "File position pos1 and pos2 are equal" << endl;

    file.open( "fpos_op_neq.txt" );
    file.seekg( 0 ); // Goes to a zero-based position in the file
    pos1 = file.tellg( );
    file.get( c );
    cout << c << endl;

    // Increment pos1
    pos1 += 1;
    file.get( c );
    cout << c << endl;

    pos1 = file.tellg( ) - fpos<int>( 2 );
    file.seekg( pos1 );
    file.get( c );
    cout << c << endl;

    // Increment pos1
    pos1 = pos1 + fpos<int>( 1 );
    file.get(c);
    cout << c << endl;

    pos1 -= fpos<int>( 2 );
    file.seekg( pos1 );
    file.get( c );
    cout << c << endl;

    file.close( );
}
```

fpos::operator+

Increments a file-position indicator.

```
fpos<Statetype> operator+(streamoff _Off) const;
```

Parameters

_Off

The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function returns **fpos(*this) +=** `_Off` .

Example

See [operator!=](#) for a sample of using `operator+` .

fpos::operator+=

Increments a file-position indicator.

```
fpos<Statetype>& operator+=(streamoff _Off);
```

Parameters

_Off

The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function adds *_Off* to the stored offset member object and then returns ***this**. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

Example

See [operator!=](#) for a sample of using `operator+=` .

fpos::operator-

Decrements a file-position indicator.

```
streamoff operator-(const fpos<Statetype>& right) const;  
  
fpos<Statetype> operator-(streamoff _Off) const;
```

Parameters

right

File position.

_Off

Stream offset.

Return Value

The first member function returns `(streamoff)*this - (streamoff) right` . The second member function returns

```
fpos(*this) -= _Off .
```

Example

See [operator!=](#) for a sample of using `operator-` .

fpos::operator-=

Decrements a file-position indicator.

```
fpos<Statetype>& operator--(streamoff _Off);
```

Parameters

_Off

Stream offset.

Return Value

The member function returns `fpos(*this) -= _Off`.

Remarks

For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

Example

See [operator!=](#) for a sample of using `operator--`.

fpos::operator==

Tests file-position indicators for equality.

```
bool operator==(const fpos<Statetype>& right) const;
```

Parameters

right

The file-position indicator against which to compare.

Return Value

true if the file-position indicators are equal; otherwise **false**.

Remarks

The member function returns `(streamoff)*this == (streamoff)right`.

Example

See [operator!=](#) for a sample of using `operator+=`.

fpos::operator streamoff

Cast object of type `fpos` to object of type `streamoff`.

```
operator streamoff() const;
```

Remarks

The member function returns the stored offset member object and any additional offset stored as part of the `fpos_t` member object.

Example

```

// fpos_op_streampos.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    streamoff s;
    ofstream file( "rdbuf.txt");

    fpos<mbstate_t> f = file.tellp( );
    // Is equivalent to ..
    // streampos f = file.tellp( );
    s = f;
    cout << s << endl;
}

```

0

fpos::seekpos

This method is used internally by the C++ Standard Library only. Do not call this method from your code.

```
fpos_t seekpos() const;
```

fpos::state

Sets or returns the conversion state.

```

Statetype state() const;

void state(Statetype _State);

```

Parameters

_State

The new conversion state.

Return Value

The conversion state.

Remarks

The first member function returns the value stored in the `st` member object. The second member function stores *_State* in the `st` member object.

Example

```
// fpos_state.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main() {
    using namespace std;
    streamoff s;
    ifstream file( "fpos_state.txt" );

    fpos<mbstate_t> f = file.tellg( );
    char ch;
    while ( !file.eof( ) )
        file.get( ch );
    s = f;
    cout << f.state( ) << endl;
    f.state( 9 );
    cout << f.state( ) << endl;
}
```

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

ios_base Class

11/8/2018 • 16 minutes to read • [Edit Online](#)

The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters. (The template class [basic_ios](#) describes what is common and is dependent on template parameters.)

An object of class `ios_base` stores formatting information, which consists of:

- Format flags in an object of type [fmtflags](#).
- An exception mask in an object of type [iostate](#).
- A field width in an object of type **int**.
- A display precision in an object of type **int**.
- A locale object in an object of type `locale`.
- Two extensible arrays, with elements of type **long** and **void** pointer.

An object of class `ios_base` also stores stream state information, in an object of type [iostate](#), and a callback stack.

Constructors

CONSTRUCTOR	DESCRIPTION
ios_base	Constructs <code>ios_base</code> objects.

Typedefs

TYPE NAME	DESCRIPTION
event_callback	Describes a function passed to register_call .
fmtflags	Constants to specify the appearance of output.
iostate	Defines constants describing the state of a stream.
openmode	Describes how to interact with a stream.
seekdir	Specifies starting point for offset operations.

Enums

event	Specifies event types.
-----------------------	------------------------

Constants

--	--

<code>adjustfield</code>	A bitmask defined as <code>internal</code> <code>left</code> <code>right</code> .
<code>app</code>	Specifies seeking to the end of a stream before each insertion.
<code>ate</code>	Specifies seeking to the end of a stream when its controlling object is first created.
<code>badbit</code>	Records a loss of integrity of the stream buffer.
<code>basefield</code>	A bitmask defined as <code>dec</code> <code>hex</code> <code>oct</code> .
<code>beg</code>	Specifies seeking relative to the beginning of a sequence.
<code>binary</code>	Specifies that a file should be read as a binary stream, rather than as a text stream.
<code>boolalpha</code>	Specifies insertion or extraction of objects of type bool as names (such as true and false) rather than as numeric values.
<code>cur</code>	Specifies seeking relative to the current position within a sequence.
<code>dec</code>	Specifies insertion or extraction of integer values in decimal format.
<code>end</code>	Specifies seeking relative to the end of a sequence.
<code>eofbit</code>	Records end-of-file while extracting from a stream.
<code>failbit</code>	Records a failure to extract a valid field from a stream.
<code>fixed</code>	Specifies insertion of floating-point values in fixed-point format (with no exponent field).
<code>floatfield</code>	A bitmask defined as <code>fixed</code> <code>scientific</code>
<code>goodbit</code>	All state bits clear.
<code>hex</code>	Specifies insertion or extraction of integer values in hexadecimal format.
<code>in</code>	Specifies extraction from a stream.
<code>internal</code>	Pads to a field width by inserting fill characters at a point internal to a generated numeric field.
<code>left</code>	Specifies left justification.
<code>oct</code>	Specifies insertion or extraction of integer values in octal format.

out	Specifies insertion to a stream.
right	Specifies right justification.
scientific	Specifies insertion of floating-point values in scientific format (with an exponent field).
showbase	Specifies insertion of a prefix that reveals the base of a generated integer field.
showpoint	Specifies unconditional insertion of a decimal point in a generated floating-point field.
showpos	Specifies insertion of a plus sign in a nonnegative generated numeric field.
skipws	Specifies skipping leading white space before certain extractions.
trunc	Specifies deleting contents of an existing file when its controlling object is created.
unitbuf	Causes output to be flushed after each insertion.
uppercase	Specifies insertion of uppercase equivalents of lowercase letters in certain insertions.

Member functions

MEMBER FUNCTION	DESCRIPTION
failure	The member class serves as the base class for all exceptions thrown by the member function clear in template class basic_ios .
flags	Sets or returns the current flag settings.
getloc	Returns the stored locale object.
imbue	Changes the locale.
Init	Creates the standard iostream objects when constructed.
iword	Assigns a value to be stored as an <code>iword</code> .
precision	Specifies the number of digits to display in a floating-point number.
pword	Assigns a value to be stored as a <code>pword</code> .
register_callback	Specifies a callback function.
setf	Sets the specified flags.

MEMBER FUNCTION	DESCRIPTION
sync_with_stdio	Ensures that iostream and C run-time library operations occur in the order that they appear in source code.
unsetf	Causes the specified flags to be off.
width	Sets the length of the output stream.
xalloc	Specifies that a variable shall be part of the stream.

Operators

OPERATOR	DESCRIPTION
operator=	The assignment operator for <code>ios_base</code> objects.

Requirements

Header: `<ios>`

Namespace: `std`

`ios_base::event`

Specifies event types.

```
enum event {
    erase_event,
    imbue_event,
    copyfmt_event};
```

Remarks

The type is an enumerated type that describes an object that can store the callback event used as an argument to a function registered with [register_callback](#). The distinct event values are:

- `copyfmt_event`, to identify a callback that occurs near the end of a call to [copyfmt](#), just before the [exception mask](#) is copied.
- `erase_event`, to identify a callback that occurs at the beginning of a call to [copyfmt](#), or at the beginning of a call to the destructor for **`*this`**.
- `imbue_event`, to identify a callback that occurs at the end of a call to [imbue](#), just before the function returns.

Example

See [register_callback](#) for an example.

`ios_base::event_callback`

Describes a function passed to [register_call](#).

```
typedef void (__cdecl *event_callback)(
    event _E,
    ios_base& _Base,
    int _I);
```

Parameters

_E

The [event](#).

_Base

The stream in which the event was called.

_I

A user-defined number.

Remarks

The type describes a pointer to a function that can be registered with [register_callback](#). This type of function must not throw an exception.

Example

See [register_call](#) for an example that uses `event_callback`.

ios_base::failure

The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the `iostreams` library, to report errors detected during stream buffer operations.

```
namespace std {
    class failure : public system_error {
    public:
        explicit failure(
            const string& _Message,
            const error_code& _Code = io_errc::stream);

        explicit failure(
            const char* str,
            const error_code& _Code = io_errc::stream);
    };
}
```

Remarks

The value returned by `what()` is a copy of `_Message`, possibly augmented with a test based on `_Code`. If `_Code` is not specified, the default value is `make_error_code(io_errc::stream)`.

Example

```
// ios_base_failure.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.exceptions(ios::failbit);
    try
    {
        file.open( "rm.txt", ios_base::in );
        // Opens nonexistent file for reading
    }
    catch( ios_base::failure f )
    {
        cout << "Caught an exception: " << f.what() << endl;
    }
}
```

```
Caught an exception: ios_base::failbit set
```

ios_base::flags

Sets or returns the current flag settings.

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

Parameters

fmtfl

The new `fmtflags` setting.

Return Value

The previous or current `fmtflags` setting.

Remarks

See [ios_base::fmtflags](#) for a list of the flags.

The first member function returns the stored format flags. The second member function stores *fmtfl* in the format flags and returns its previous stored value.

Example

```
// ios_base_flags.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    cout << cout.flags( ) << endl;
    cout.flags( ios::dec | ios::boolalpha );
    cout << cout.flags( );
}
```

ios_base::fmtflags

Constants to specify the appearance of output.

```
class ios_base {
public:
    typedef implementation-defined-bitmask-type fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags hex;
    static const fmtflags internal;
    static const fmtflags left;
    static const fmtflags oct;
    static const fmtflags right;
    static const fmtflags scientific;
    static const fmtflags showbase;
    static const fmtflags showpoint;
    static const fmtflags showpos;
    static const fmtflags skipws;
    static const fmtflags unitbuf;
    static const fmtflags uppercase;
    static const fmtflags adjustfield;
    static const fmtflags basefield;
    static const fmtflags floatfield;
    // ...
};
```

Remarks

Supports the manipulators in [ios](#).

The type is a bitmask type that describes an object that can store format flags. The distinct flag values (elements) are:

- `dec`, to insert or extract integer values in decimal format.
- `hex`, to insert or extract integer values in hexadecimal format.
- `oct`, to insert or extract integer values in octal format.
- `showbase`, to insert a prefix that reveals the base of a generated integer field.
- `internal`, to pad to a field width as needed by inserting fill characters at a point internal to a generated numeric field. (For information on setting the field width, see [setw](#)).
- `left`, to pad to a field width as needed by inserting fill characters at the end of a generated field (left justification).
- `right`, to pad to a field width as needed by inserting fill characters at the beginning of a generated field (right justification).
- `boolalpha`, to insert or extract objects of type **bool** as names (such as **true** and **false**) rather than as numeric values.
- `fixed`, to insert floating-point values in fixed-point format (with no exponent field).
- `scientific`, to insert floating-point values in scientific format (with an exponent field).

- `showpoint`, to insert a decimal point unconditionally in a generated floating-point field.
- `showpos`, to insert a plus sign in a nonnegative generated numeric field.
- `skipws`, to skip leading white space before certain extractions.
- `unitbuf`, to flush output after each insertion.
- `uppercase`, to insert uppercase equivalents of lowercase letters in certain insertions.

In addition, several useful values are:

- `adjustfield`, a bitmask defined as `internal` | `left` | `right`
- `basefield`, defined as `dec` | `hex` | `oct`
- `floatfield`, defined as `fixed` | `scientific`

For examples of functions that modify these format flags, see [<iomanip>](#).

ios_base::getloc

Returns the stored locale object.

```
locale getloc() const;
```

Return Value

The stored locale object.

Example

```
// ios_base_getlock.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << cout.getloc( ).name( ).c_str( ) << endl;
}
```

C

ios_base::imbue

Changes the locale.

```
locale imbue(const locale& _Loc);
```

Parameters

_Loc

The new locale setting.

Return Value

The previous locale.

Remarks

The member function stores `_Loc` in the locale object and then reports the callback event and `imbue_event`. It returns the previous stored value.

Example

See [basic_ios::imbue](#) for a sample.

ios_base::Init

Creates the standard iostream objects when constructed.

```
class Init { };
```

Remarks

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed, even before the execution of a constructor for an arbitrary static object.

ios_base::ios_base

Constructs `ios_base` objects.

```
ios_base();
```

Remarks

The (protected) constructor does nothing. A later call to **`basic_ios::init`** must initialize the object before it can be safely destroyed. Thus, the only safe use for class `ios_base` is as a base class for template class [basic_ios](#).

ios_base::iostate

The type of constants that describe the state of a stream.

```
class ios_base {
public:
    typedef implementation-defined-bitmask-type iostate;
    static const iostate badbit;
    static const iostate eofbit;
    static const iostate failbit;
    static const iostate goodbit;
    // ...
};
```

Remarks

The type is a bitmask type that describes an object that can store stream state information. The distinct flag values (elements) are:

- `badbit`, to record a loss of integrity of the stream buffer.
- `eofbit`, to record end-of-file while extracting from a stream.
- `failbit`, to record a failure to extract a valid field from a stream.

In addition, a useful value is `goodbit`, where none of the previously mentioned bits are set (`goodbit` is guaranteed to be zero).

ios_base::iword

Assigns a value to be stored as an `iword`.

```
long& iword(int idx);
```

Parameters

idx

The index of the value to store as an `iword`.

Remarks

The member function returns a reference to element *idx* of the extensible array with elements of type **long**. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to `iword` for the object, after the object is altered by a call to **basic_ios::copyfmt**, or after the object is destroyed.

If *idx* is negative or if unique storage is unavailable for the element, the function calls **setstate(badbit)** and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call **xalloc**.

Example

See **xalloc** for a sample of how to use `iword`.

ios_base::openmode

Describes how to interact with a stream.

```
class ios_base {
public:
    typedef implementation-defined-bitmask-type iostate;
    static const iostate badbit;
    static const iostate eofbit;
    static const iostate failbit;
    static const iostate goodbit;
    // ...
};
```

Remarks

The type is a `bitmask type` that describes an object that can store the opening mode for several iostreams objects. The distinct flag values (elements) are:

- `app`, to seek to the end of a stream before each insertion.
- `ate`, to seek to the end of a stream when its controlling object is first created.
- `binary`, to read a file as a binary stream, rather than as a text stream.
- `in`, to permit extraction from a stream.
- `out`, to permit insertion to a stream.
- `trunc`, to delete contents of an existing file when its controlling object is created.

Example

```
// ios_base_openmode.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.open( "rm.txt", ios_base::out | ios_base::trunc );

    file << "testing";
}
```

ios_base::operator=

The assignment operator for ios_base objects.

```
ios_base& operator=(const ios_base& right);
```

Parameters

right

An object of type `ios_base`.

Return Value

The object being assigned to.

Remarks

The operator copies the stored formatting information, making a new copy of any extensible arrays. It then returns ***this**. Note that the callback stack is not copied.

This operator is only used by classes derived from `ios_base`.

ios_base::precision

Specifies the number of digits to display in a floating-point number.

```
streamsize precision() const;
streamsize precision(streamsize _Prec);
```

Parameters

_Prec

The number of significant digits to display, or the number of digits after the decimal point in fixed notation.

Return Value

The first member function returns the stored [display precision](#). The second member function stores *_Prec* in the display precision and returns its previous stored value.

Remarks

Floating-point numbers are displayed in fixed notation with [fixed](#).

Example


```
// ios_base_precision.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    float i = 31.31234F;

    cout.precision( 3 );
    cout << i << endl;          // display three significant digits
    cout << fixed << i << endl; // display three digits after decimal
                                // point
}
```

```
31.3
31.312
```

ios_base::pword

Assigns a value to be stored as a `pword`.

```
void *& pword(int _Idx);
```

Parameters

_Idx

The index of the value to store as a `pword`.

Remarks

The member function returns a reference to element *_Idx* of the extensible array with elements of type **void** pointer. All elements are effectively present and initially store the null pointer. The returned reference is invalid after the next call to `pword` for the object, after the object is altered by a call to **basic_ios::copyfmt**, or after the object is destroyed.

If *_Idx* is negative, or if unique storage is unavailable for the element, the function calls **setstate(badbit)** and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call **xalloc**.

Example

See **xalloc** for an example of using `pword`.

ios_base::register_callback

Specifies a callback function.

```
void register_callback(
    event_callback pfn, int idx);
```

Parameters

pfn

Pointer to the callback function.

idx

A user-defined number.

Remarks

The member function pushes the pair `{pfn, idx}` onto the stored callback stack [callback stack](#). When a callback event **ev** is reported, the functions are called, in reverse order of registry, by the expression

```
(*pfn)(ev, *this, idx).
```

Example

```
// ios_base_register_callback.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

void callback1( ios_base::event e, ios_base& stream, int arg )
{
    cout << "in callback1" << endl;
    switch ( e )
    {
        case ios_base::erase_event:
            cout << "an erase event" << endl;
            break;
        case ios_base::imbue_event:
            cout << "an imbue event" << endl;
            break;
        case ios_base::copyfmt_event:
            cout << "an copyfmt event" << endl;
            break;
    };
}

void callback2( ios_base::event e, ios_base& stream, int arg )
{
    cout << "in callback2" << endl;
    switch ( e )
    {
        case ios_base::erase_event:
            cout << "an erase event" << endl;
            break;
        case ios_base::imbue_event:
            cout << "an imbue event" << endl;
            break;
        case ios_base::copyfmt_event:
            cout << "an copyfmt event" << endl;
            break;
    };
}

int main( )
{
    // Make sure the imbue will not throw an exception
    // assert( setlocale( LC_ALL, "german" )!=NULL );

    cout.register_callback( callback1, 0 );
    cin.register_callback( callback2, 0 );

    try
    {
        // If no exception because the locale's not found,
        // generate an imbue_event on callback1
        cout.imbue(locale("german"));
    }
    catch(...)
    {
        cout << "exception" << endl;
    }
}
```

```

    }

    // This will
    // (1) erase_event on callback1
    // (2) copyfmt_event on callback2
    cout.copyfmt(cin);

    // We get two erase events from callback2 at the end because
    // both cin and cout have callback2 registered when cin and cout
    // are destroyed at the end of program.
}

```

```

in callback1
an imbue event
in callback1
an erase event
in callback2
an copyfmt event
in callback2
an erase event
in callback2
an erase event

```

ios_base::seekdir

Specifies starting point for offset operations.

```

namespace std {
    class ios_base {
    public:
        typedef implementation-defined-enumerated-type seekdir;
        static const seekdir beg;
        static const seekdir cur;
        static const seekdir end;
        // ...
    };
}

```

Remarks

The type is an enumerated type that describes an object that can store the seek mode used as an argument to the member functions of several iostream classes. The distinct flag values are:

- `beg`, to seek (alter the current read or write position) relative to the beginning of a sequence (array, stream, or file).
- `cur`, to seek relative to the current position within a sequence.
- `end`, to seek relative to the end of a sequence.

Example

```
// ios_base_seekdir.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.open( "rm.txt", ios_base::out | ios_base::trunc );

    file << "testing";
    file.seekp( 0, ios_base::beg );
    file << "a";
    file.seekp( 0, ios_base::end );
    file << "a";
}
```

ios_base::setf

Sets the specified flags.

```
fmtflags setf(
    fmtflags _Mask
);
fmtflags setf(
    fmtflags _Mask,
    fmtflags _Unset
);
```

Parameters

_Mask

The flags to turn on.

_Unset

The flags to turn off.

Return Value

The previous format flags

Remarks

The first member function effectively calls `flags(_Mask | _Flags)` (set selected bits) and then returns the previous format flags. The second member function effectively calls `flags(_Mask & fmtfl, flags & ~_Mask)` (replace selected bits under a mask) and then returns the previous format flags.

Example

```
// ios_base_setf.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int i = 10;
    cout << i << endl;

    cout.unsetf( ios_base::dec );
    cout.setf( ios_base::hex );
    cout << i << endl;

    cout.setf( ios_base::dec );
    cout << i << endl;
    cout.setf( ios_base::hex, ios_base::dec );
    cout << i << endl;
}
```

ios_base::sync_with_stdio

Ensures that iostream and C run-time library operations occur in the order that they appear in source code.

```
static bool sync_with_stdio(
    bool _Sync = true
);
```

Parameters

_Sync

Whether all streams are in sync with `stdio`.

Return Value

Previous setting for this function.

Remarks

The static member function stores a `stdio` sync flag, which is initially **true**. When **true**, this flag ensures that operations on the same file are properly synchronized between the [iostreams](#) functions and those defined in the C++ Standard Library. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores *_Sync* in the `stdio` sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

ios_base::unsetf

Causes the specified flags to be off.

```
void unsetf(
    fmtflags _Mask
);
```

Parameters

_Mask

The flags that you want off.

Remarks

The member function effectively calls `flags(~_Mask & flags)` (clear selected bits).

Example

See `ios_base::setf` for a sample of using `unsetf`.

ios_base::width

Sets the length of the output stream.

```
streamsize width( ) const;
streamsize width(
    streamsize _Wide
);
```

Parameters

_Wide

The desired size of the output stream.

Return Value

The current width setting.

Remarks

The first member function returns the stored field width. The second member function stores *_Wide* in the field width and returns its previous stored value.

Example

```
// ios_base_width.cpp
// compile with: /EHsc
#include <iostream>

int main( ) {
    using namespace std;

    cout.width( 20 );
    cout << cout.width( ) << endl;
    cout << cout.width( ) << endl;
}
```

```
20
0
```

ios_base::xalloc

Specifies that a variable is part of the stream.

```
static int xalloc( );
```

Return Value

The static member function returns a stored static value, which it increments on each call.

Remarks

You can use the return value as a unique index argument when calling the member functions `word` or `pword`.

Example

```
// ios_base_xalloc.cpp
// compile with: /EHsc
// Lets you store user-defined information.
// iword, jword, xalloc
#include <iostream>

int main( )
{
    using namespace std;

    static const int i = ios_base::xalloc();
    static const int j = ios_base::xalloc();
    cout.iword( i ) = 11;
    cin.iword( i ) = 13;
    cin.pword( j ) = "testing";
    cout << cout.iword( i ) << endl;
    cout << cin.iword( i ) << endl;
    cout << ( char * )cin.pword( j ) << endl;
}
```

```
11
13
testing
```

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<iosfwd>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Declares forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of its declarations, but not its definition.

Syntax

```
#include <iosfwd>
```

Typedefs

```
typedef T1 streamoff;
typedef T2 streamsize;
typedef fpos streampos;

// wchar_t TYPE DEFINITIONS
typedef basic_ios<char, char_traits<char>> ios;
typedef basic_streambuf<char, char_traits<char>> streambuf;
typedef basic_istream<char, char_traits<char>> istream;
typedef basic_ostream<char, char_traits<char>> ostream;
typedef basic_iostream<char, char_traits<char>> iostream;
typedef basic_stringbuf<char, char_traits<char>> stringbuf;
typedef basic_istreambuf<char, char_traits<char>> istreambuf;
typedef basic_ostreambuf<char, char_traits<char>> ostreambuf;
typedef basic_stringstream<char, char_traits<char>> stringstream;
typedef basic_filebuf<char, char_traits<char>> filebuf;
typedef basic_ifstream<char, char_traits<char>> ifstream;
typedef basic_ofstream<char, char_traits<char>> ofstream;
typedef basic_fstream<char, char_traits<char>> fstream;

// wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t>> wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t>> wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t>> wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t>> wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t>> wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t>> wstringbuf;
typedef basic_istreambuf<wchar_t, char_traits<wchar_t>> wistreambuf;
typedef basic_ostreambuf<wchar_t, char_traits<wchar_t>> wostreambuf;
typedef basic_stringstream<wchar_t, char_traits<wchar_t>> wstringstream;
typedef basic_filebuf<wchar_t, char_traits<wchar_t>> wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t>> wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t>> wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t>> wfstream;
};
```

Forward Declarations/Template Classes


```

template <class _Statetype>
class fpos;

template <class Elem>;
class char_traits;

class char_traits<char>;

class char_traits<wchar_t>;

template <class T>
class allocator;

class ios_base;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ios;

template <class Elem, class Tr = char_traits<Elem>>
class istreambuf_iterator;

template <class Elem, class Tr = char_traits<Elem>>
class ostreambuf_iterator;

template <class Elem, class Tr = char_traits<Elem>>
class basic_streambuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_istream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ostream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_iostream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_stringbuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_istreamstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ostreamstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_stringstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_filebuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ifstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ofstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_fstream;

```

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

<iostream>

11/9/2018 • 4 minutes to read • [Edit Online](#)

Declares objects that control reading from and writing to the standard streams. This is often the only header you need to include to perform input and output from a C++ program.

Syntax

```
#include <iostream>
```

Remarks

The objects fall into two groups:

- `cin`, `cout`, `cerr`, and `clog` are byte oriented, performing conventional byte-at-a-time transfers.
- `wcin`, `wcout`, `wcerr`, and `wclog` are wide oriented, translating to and from the wide characters that the program manipulates internally.

Once you perform certain operations on a stream, such as the standard input, you cannot perform operations of a different orientation on the same stream. Therefore, a program cannot operate interchangeably on both `cin` and `wcin`, for example.

All the objects declared in this header share a peculiar property — you can assume they are constructed before any static objects you define, in a translation unit that includes `<iostream>`. Equally, you can assume that these objects are not destroyed before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Therefore, you can safely read from or write to the standard streams before program startup and after program termination.

This guarantee is not universal, however. A static constructor may call a function in another translation unit. The called function cannot assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class `ios_base::Init`.

Global Stream Objects

<code>cerr</code>	Specifies the <code>cerr</code> global stream.
<code>cin</code>	Specifies the <code>cin</code> global stream.
<code>clog</code>	Specifies the <code>clog</code> global stream.
<code>cout</code>	Specifies the <code>cout</code> global stream.
<code>wcerr</code>	Specifies the <code>wcerr</code> global stream.
<code>wcin</code>	Specifies the <code>wcin</code> global stream.

<code>wclog</code>	Specifies the <code>wclog</code> global stream.
<code>wcout</code>	Specifies the <code>wcout</code> global stream.

cerr

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<stdio>`.

```
extern ostream cerr;
```

Return Value

An `ostream` object.

Remarks

The object controls unbuffered insertions to the standard error output as a byte stream. Once the object is constructed, the expression `cerr.flags() & unitbuf` is nonzero, and `cerr.tie() == &cout`.

Example

```
// iostream_cerr.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

void TestWide( )
{
    int i = 0;
    wcout << L"Enter a number: ";
    wcin >> i;
    wcerr << L"test for wcerr" << endl;
    wclog << L"test for wclog" << endl;
}

int main( )
{
    int i = 0;
    cout << "Enter a number: ";
    cin >> i;
    cerr << "test for cerr" << endl;
    clog << "test for clog" << endl;
    TestWide( );
}
```

cin

Specifies the `cin` global stream.

```
extern istream cin;
```

Return Value

An `istream` object.

Remarks

The object controls extractions from the standard input as a byte stream. Once the object is constructed, the call `cin.tie()` returns `&cout`.

Example

In this example, `cin` sets the fail bit on the stream when it encounters non-numeric characters. The program clears the fail bit and strips the invalid character from the stream to proceed.

```
// iostream_cin.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
    int x;
    cout << "enter choice:";
    cin >> x;
    while (x < 1 || x > 4)
    {
        cout << "Invalid choice, try again:";
        cin >> x;
        // not a numeric character, probably
        // clear the failure and pull off the non-numeric character
        if (cin.fail())
        {
            cin.clear();
            char c;
            cin >> c;
        }
    }
}
```

2

clog

Specifies the `clog` global stream.

```
extern ostream clog;
```

Return Value

An [ostream](#) object.

Remarks

The object controls buffered insertions to the standard error output as a byte stream.

Example

See [cerr](#) for an example of using `clog`.

cout

Specifies the `cout` global stream.

```
extern ostream cout;
```

Return Value

An [ostream](#) object.

Remarks

The object controls insertions to the standard output as a byte stream.

Example

See [cerr](#) for an example of using `cout`.

wcerr

Specifies the `wcerr` global stream.

```
extern wostream wcerr;
```

Return Value

A [wostream](#) object.

Remarks

The object controls unbuffered insertions to the standard error output as a wide stream. Once the object is constructed, the expression `wcerr.flags() & unitbuf` is nonzero.

Example

See [cerr](#) for an example of using `wcerr`.

wcin

Specifies the `wcin` global stream.

```
extern wistream wcin;
```

Return Value

A [wistream](#) object.

Remarks

The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call `wcin.tie()` returns `&wcout`.

Example

See [cerr](#) for an example of using `wcin`.

wclog

Specifies the `wclog` global stream.

```
extern wostream wclog;
```

Return Value

A [wostream](#) object.

Remarks

The object controls buffered insertions to the standard error output as a wide stream.

Example

See [cerr](#) for an example of using `wclog`.

wcout

Specifies the `wcout` global stream.

```
extern wostream wcout;
```

Return Value

A [wostream](#) object.

Remarks

The object controls insertions to the standard output as a wide stream.

Example

See [cerr](#) for an example of using `wcout`.

`CString` instances in a `wcout` statement must be cast to `const wchar_t*`, as shown in the following example.

```
CString cs("meow");  
  
wcout <<(const wchar_t*) cs <<endl;
```

For more information, see [Basic CString Operations](#).

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<iostream>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template class `basic_istream`, which mediates extractions for the iostreams, and the template class `basic_iostream`, which mediates both insertions and extractions. The header also defines a related manipulator. This header file is typically included for you by another iostreams header; you rarely have to include it directly.

Syntax

```
#include <iostream>
```

Typedefs

TYPE NAME	DESCRIPTION
<code>iostream</code>	A type <code>basic_iostream</code> specialized on char .
<code>istream</code>	A type <code>basic_istream</code> specialized on char .
<code>wiostream</code>	A type <code>basic_iostream</code> specialized on wchar .
<code>wistream</code>	A type <code>basic_istream</code> specialized on wchar .

Manipulators

<code>ws</code>	Skips white space in the stream.
<code>swap</code>	Exchanges two stream objects.

Operators

OPERATOR	DESCRIPTION
<code>operator>></code>	Extracts characters and strings from the stream.

Classes

CLASS	DESCRIPTION
<code>basic_iostream</code>	A stream class that can do both input and output.
<code>basic_istream</code>	The template class describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type <code>Elem</code> , also known as <code>char_type</code> , whose character traits are determined by the class <code>Tr</code> , also known as <code>traits_type</code> .

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<istream> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

ws

swap

Exchanges the elements of two stream objects.

```
template <class Elem, class Tr>
void swap(
    basic_istream<Elem, Tr>& left,
    basic_istream<Elem, Tr>& right);

template <class Elem, class Tr>
void swap(
    basic_iostream<Elem, Tr>& left,
    basic_iostream<Elem, Tr>& right);
```

Parameters

left

A stream.

right

A stream.

WS

Skips white space in the stream.

```
template class<Elem, Tr> basic_istream<Elem, Tr>& ws(basic_istream<Elem, Tr>& _Istr);
```

Parameters

_Istr

A stream.

Return Value

The stream.

Remarks

The manipulator extracts and discards any elements `ch` for which `use_facet< ctype< Elem> >(_Istr).is(ctype< Elem>::space, ch)` is true.

The function calls `setstate(eofbit)` if it encounters end of file while extracting elements. It returns *_Istr*.

Example

See [operator>>](#) for an example of using `ws`.

See also

[<istream>](#)

<istream> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

operator>>

Extracts characters and strings from the stream.

```
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr,
    Elem* str);

template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr,
    Elem& Ch);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    signed char* str);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    signed char& Ch);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    unsigned char* str);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    unsigned char& Ch);

template <class Elem, class Tr, class Type>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<char, Tr>&& Istr,
    Type& val);
```

Parameters

Ch

A character.

Istr

A stream.

str

A string.

val

A type.

Return Value

The stream

Remarks

The `basic_istream` class also defines several extraction operators. For more information, see [basic_istream::operator>>](#).

The template function:

```
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr, Elem* str);
```

extracts up to $N - 1$ elements and stores them in the array starting at `_Str`. If `Istr.width` is greater than zero, N is `Istr.width`; otherwise, it is the size of the largest array of `Elem` that can be declared. The function always stores the value `Elem()` after any extracted elements it stores. Extraction stops early on end of file, on a character with value `Elem(0)` (which is not extracted), or on any element (which is not extracted) that would be discarded by [ws](#). If the function extracts no elements, it calls `Istr.setstate(failbit)`. In any case, it calls `Istr.width(0)` and returns `Istr`.

Security Note The null-terminated string being extracted from the input stream must not exceed the size of the destination buffer `str`. For more information, see [Avoiding Buffer Overruns](#).

The template function:

```
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr, Elem& Ch);
```

extracts an element, if it is possible, and stores it in `Ch`. Otherwise, it calls `is.setstate(failbit)`. In any case, it returns `Istr`.

The template function:

```
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, signed char* str);
```

returns `Istr >> (char *) str`.

The template function:

```
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, signed char& Ch);
```

returns `Istr >> (char&) Ch`.

The template function:

```
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, unsigned char* str);
```

returns `Istr >> (char *) str`.

The template function:

```
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, unsigned char& Ch);
```

returns `Istr >> (char&) Ch` .

The template function:

```
template <class Elem, class Tr, class Type>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<char, Tr>&& Istr,
    Type& val);
```

returns `Istr >> val` (and converts an rvalue reference to `Istr` to an lvalue in the process).

Example

```
// istream_op_extract.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    ws( cin );
    char c[10];

    cin.width( 9 );
    cin >> c;
    cout << c << endl;
}
```

See also

[<istream>](#)

<istream> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

iostream	istream	wiostream
wistream		

iostream

A type `basic_iostream` specialized on **char**.

```
typedef basic_iostream<char, char_traits<char>> iostream;
```

Remarks

The type is a synonym for template class [basic_iostream](#), specialized for elements of type **char** with default character traits.

istream

A type `basic_istream` specialized on **char**.

```
typedef basic_istream<char, char_traits<char>> istream;
```

Remarks

The type is a synonym for template class [basic_istream](#), specialized for elements of type **char** with default character traits.

wiostream

A type `basic_iostream` specialized on **wchar_t**.

```
typedef basic_iostream<wchar_t, char_traits<wchar_t>> wiostream;
```

Remarks

The type is a synonym for template class [basic_iostream](#), specialized for elements of type **wchar_t** with default character traits.

wistream

A type `basic_istream` specialized on **wchar_t**.

```
typedef basic_istream<wchar_t, char_traits<wchar_t>> wistream;
```

Remarks

The type is a synonym for template class [basic_istream](#), specialized for elements of type **wchar_t** with default character traits.

See also

[<istream>](#)

basic_iostream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A stream class that can do both input and output.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_iostream : public basic_istream<Elem, Tr>,
    public basic_ostream<Elem, Tr>
{
public:
    explicit basic_iostream(basic_streambuf<Elem, Tr>* strbuf);

    virtual ~basic_iostream();

};
```

Remarks

The template class describes an object that controls insertions, through its base class `basic_ostream` < `Elem`, `Tr` >, and extractions, through its base class `basic_istream` < `Elem`, `Tr` >. The two objects share a common virtual base class `basic_ios` < `Elem`, `Tr` >. They also manage a common stream buffer, with elements of type `Elem`, whose character traits are determined by the class `Tr`. The constructor initializes its base classes through `basic_istream` (`strbuf`) and `basic_ostream` (`strbuf`).

Constructors

CONSTRUCTOR	DESCRIPTION
<code>basic_iostream</code>	Create a <code>basic_iostream</code> object.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>swap</code>	Exchanges the contents of the provided <code>basic_iostream</code> object for the contents of this object.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns the value of a specified <code>basic_iostream</code> object to this object. This is a move assignment involving an <code>rvalue</code> that does not leave a copy behind.

Requirements

Header: <istream>

Namespace: std

basic_iostream::basic_iostream

Create a `basic_iostream` object.

```
explicit basic_iostream(basic_streambuf<Elem, Tr>* strbuf);

basic_iostream(basic_iostream&& right);

basic_iostream();
```

Parameters

strbuf

An existing `basic_streambuf` object.

right

An existing `basic_iostream` object that is used to construct a new `basic_iostream`.

Remarks

The first constructor initializes the base objects by way of `basic_istream(strbuf)` and `basic_ostream(strbuf)`.

The second constructor initializes the base objects by calling `move(right)`.

basic_iostream::operator=

Assign the value of a specified `basic_iostream` object to this object. This is a move assignment involving an rvalue that does not leave a copy behind.

```
basic_iostream& operator=(basic_iostream&& right);
```

Parameters

right

An `rvalue` reference to a `basic_iostream` object to assign from.

Remarks

The member operator calls `swap(right)`.

basic_iostream::swap

Exchanges the contents of the provided `basic_iostream` object for the contents of this object.

```
void swap(basic_iostream& right);
```

Parameters

right

The `basic_iostream` object to swap.

Remarks

The member function calls `swap(right)`.

See also

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

basic_istream Class

11/9/2018 • 15 minutes to read • [Edit Online](#)

Describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type `Elem`, also known as `char_type`, whose character traits are determined by the class `Tr`, also known as `traits_type`.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_istream : virtual public basic_ios<Elem, Tr>
```

Remarks

Most of the member functions that overload `operator>>` are formatted input functions. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);

if (ok)
{
    try
    {
        /*extract elements and convert
        accumulate flags in state.
        store a successful conversion*/
    }
    catch (...)
    {
        try
        {
            setstate(badbit);

        }
        catch (...)
        {
        }
        if ((exceptions() & badbit) != 0)
            throw;
    }
}
setstate(state);

return (*this);
```

Many other member functions are unformatted input functions. They follow the pattern:

```

iostate state = goodbit;
count = 0;    // the value returned by gcount
const sentry ok(*this, true);

if (ok)
{
    try
    {
        /* extract elements and deliver
           count extracted elements in count
           accumulate flags in state */
    }
    catch (...)
    {
        try
        {
            setstate(badbit);

        }
        catch (...)
        {
        }
        if ((exceptions() & badbit) != 0)
            throw;
    }
}
setstate(state);

```

Both groups of functions call `setstate(eofbit)` if they encounter end of file while extracting elements.

An object of class `basic_istream < Elem, Tr >` stores:

- A virtual public base object of class `basic_ios < Elem, Tr >`.
- An extraction count for the last unformatted input operation (called `count` in the previous code).

Example

See the example for [basic_ifstream Class](#) to learn more about input streams.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_istream	Constructs an object of type <code>basic_istream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
gcount	Returns the number of characters read during the last unformatted input.
get	Reads one or more characters from the input stream.
getline	Reads a line from the input stream.
ignore	Causes a number of elements to be skipped from the current read position.

MEMBER FUNCTION	DESCRIPTION
peek	Returns the next character to be read.
putback	Puts a specified character into the stream.
read	Reads a specified number of characters from the stream and stores them in an array.
readsome	Read from buffer only.
seekg	Moves the read position in a stream.
sentry	The nested class describes an object whose declaration structures the formatted input functions and the unformatted input functions.
swap	Exchanges this <code>basic_istream</code> object for the provided <code>basic_istream</code> object parameter.
sync	Synchronizes the input device associated with the stream with the stream's buffer.
tellg	Reports the current read position in the stream.
unget	Puts the most recently read character back into the stream.

Operators

OPERATOR	DESCRIPTION
operator>>	Calls a function on the input stream or reads formatted data from the input stream.
operator=	Assigns the <code>basic_istream</code> on the right side of the operator to this object. This is a move assignment involving an <code>rvalue</code> reference that does not leave a copy behind.

Requirements

Header: `<istream>`

Namespace: `std`

`basic_istream::basic_istream`

Constructs an object of type `basic_istream`.

```
explicit basic_istream(
    basic_streambuf<Elem, Tr>* strbuf,
    bool _Isstd = false);

basic_istream(basic_istream&& right);
```

Parameters

strbuf

An object of type [basic_streambuf](#).

_Isstd

true if this is a standard stream; otherwise, **false**.

right

A `basic_istream` object to copy.

Remarks

The first constructor initializes the base class by calling `init(_S trbuf)`. It also stores zero in the extraction count. For more information about this extraction count, see the Remarks section of the [basic_istream Class](#) overview topic.

The second constructor initializes the base class by calling `move(right)`. It also stores `_R ight.gcount()` in the extraction count and stores zero in the extraction count for `_R ight`.

Example

See the example for [basic_ifstream::basic_ifstream](#) to learn more about input streams.

basic_istream::gcount

Returns the number of characters read during the last unformatted input.

```
streamsize gcount() const;
```

Return Value

The extraction count.

Remarks

Use [basic_istream::get](#) to read unformatted characters.

Example

```
// basic_istream_gcount.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    cout << "Type the letter 'a': ";

    ws( cin );
    char c[10];

    cin.get( &c[0],9 );
    cout << c << endl;

    cout << cin.gcount( ) << endl;
}
```

```
a
```

Type the letter 'a': a
1

basic_istream::get

Reads one or more characters from the input stream.

```
int_type get();

basic_istream<Elem, Tr>& get(Elem& Ch);
basic_istream<Elem, Tr>& get(Elem* str, streamsize count);
basic_istream<Elem, Tr>& get(Elem* str, streamsize count, Elem Delim);

basic_istream<Elem, Tr>& get(basic_streambuf<Elem, Tr>& strbuf);
basic_istream<Elem, Tr>& get(basic_streambuf<Elem, Tr>& strbuf, Elem Delim);
```

Parameters

count

The number of characters to read from `strbuf`.

Delim

The character that should terminate the read if it is encountered before *count*.

str

A string in which to write.

Ch

A character to get.

strbuf

A buffer in which to write.

Return Value

The parameterless form of `get` returns the element read as an integer or end of file. The remaining forms return the stream (`*this`).

Remarks

The first of these unformatted input functions extracts an element, if possible, as if by returning `rddbuf` -> `sbumpc`. Otherwise, it returns **traits_type::eof**. If the function extracts no element, it calls `setstate(failbit)`.

The second function extracts the `int_type` element `meta` the same way. If `meta` compares equal to **traits_type::eof**, the function calls `setstate(failbit)`. Otherwise, it stores **traits_type::to_char_type**(`meta`) in `ch`. The function returns ***this**.

The third function returns `get(_Str, count, widen('\n'))`.

The fourth function extracts up to `count - 1` elements and stores them in the array beginning at `_Str`. It always stores `char_type` after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *Delim*, in which case the element is put back to the controlled sequence.
- After the function extracts *count* - 1 elements.

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns ***this**.

The fifth function returns **get(*strbuf*, *widen* ('\\n'))**.

The sixth function extracts elements and inserts them in *strbuf*. Extraction stops on end-of-file or on an element that compares equal to *_ Delim*, which is not extracted. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls *setstate* (**failbit**). In any case, the function returns ***this**.

Example

```
// basic_istream_get.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char c[10];

    c[0] = cin.get( );
    cin.get( c[1] );
    cin.get( &c[2],3 );
    cin.get( &c[4], 4, '7' );

    cout << c << endl;
}
```

1111

basic_istream::getline

Gets a line from the input stream.

```
basic_istream<Elem, Tr>& getline(
    char_type* str,
    streamsize count);

basic_istream<Elem, Tr>& getline(
    char_type* str,
    streamsize count,
    char_type Delim);
```

Parameters

count

The number of characters to read from *strbuf*.

Delim

The character that should terminate the read if it is encountered before *count*.

str

A string in which to write.

Return Value

The stream (***this**).

Remarks

The first of these unformatted input functions returns **getline(_ Str, *count*, *widen* ('\\n'))**.

The second function extracts up to *count* - 1 elements and stores them in the array beginning at *_ Str*. It always

stores the string termination character after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *Delim*, in which case the element is neither put back nor appended to the controlled sequence.
- After the function extracts *count* - 1 elements.

If the function extracts no elements or *count* - 1 elements, it calls `setstate(failbit)`. In any case, it returns ***this**.

Example

```
// basic_istream_getline.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char c[10];

    cin.getline( &c[0], 5, '2' );
    cout << c << endl;
}
```

121

basic_istream::ignore

Causes a number of elements to be skipped from the current read position.

```
basic_istream<Elem, Tr>& ignore(
    streamsize count = 1,
    int_type Delim = traits_type::eof());
```

Parameters

count

The number of elements to skip from the current read position.

Delim

The element that, if encountered before *count*, causes `ignore` to return and allowing all elements after *Delim* to be read.

Return Value

The stream (***this**).

Remarks

The unformatted input function extracts up to *count* elements and discards them. If *count* equals **numeric_limits<int>::max**, however, it is taken as arbitrarily large. Extraction stops early on end of file or on an element `ch` such that `traits_type::to_int_type(ch)` compares equal to *Delim* (which is also extracted). The function returns ***this**.

Example

```
// basic_istream_ignore.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
    using namespace std;
    char chararray[10];
    cout << "Type 'abcdef': ";
    cin.ignore( 5, 'c' );
    cin >> chararray;
    cout << chararray;
}
```

```
Type 'abcdef': abcdef
def
```

basic_istream::operator>>

Calls a function on the input stream or reads formatted data from the input stream.

```
basic_istream& operator>>(basic_istream& (* Pfn)(basic_istream&));
basic_istream& operator>>(ios_base& (* Pfn)(ios_base&));
basic_istream& operator>>(basic_ios<Elem, Tr>& (* Pfn)(basic_ios<Elem, Tr>&));
basic_istream& operator>>(basic_streambuf<Elem, Tr>* strbuf);
basic_istream& operator>>(bool& val);
basic_istream& operator>>(short& val);
basic_istream& operator>>(unsigned short& val);
basic_istream& operator>>(int& val);
basic_istream& operator>>(unsigned int& val);
basic_istream& operator>>(long& val);
basic_istream& operator>>(unsigned long& val);
basic_istream& operator>>(long long& val);
basic_istream& operator>>(unsigned long long& val);
basic_istream& operator>>(void *& val);
basic_istream& operator>>(float& val);
basic_istream& operator>>(double& val);
basic_istream& operator>>(long double& val);
```

Parameters

Pfn

A function pointer.

strbuf

An object of type `stream_buf`.

val

The value to read from the stream.

Return Value

The stream (***this**).

Remarks

The `<istream>` header also defines several global extraction operators. For more information, see [operator>> \(<istream>\)](#).

The first member function ensures that an expression of the form **istr** >> `ws` calls `ws(istr)`, and then returns ***this**. The second and third functions ensure that other manipulators, such as `hex`, behave similarly. The remaining functions constitute the formatted input functions.

The function:

```
basic_istream& operator>>(
    basic_streambuf<Elem, Tr>* strbuf);
```

extracts elements, if `_Strbuf` is not a null pointer, and inserts them in `strbuf`. Extraction stops on end of file. It also stops without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns ***this**.

The function:

```
basic_istream& operator>>(bool& val);
```

extracts a field and converts it to a Boolean value by calling `use_facet < num_get < Elem, InIt>(getloc). get(InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator < Elem, Tr>`. The function returns ***this**.

The functions:

```
basic_istream& operator>>(short& val);
basic_istream& operator>>(unsigned short& val);
basic_istream& operator>>(int& val);
basic_istream& operator>>(unsigned int& val);
basic_istream& operator>>(long& val);
basic_istream& operator>>(unsigned long& val);
basic_istream& operator>>(long long& val);
basic_istream& operator>>(unsigned long long& val);
basic_istream& operator>>(void *& val);
```

each extract a field and convert it to a numeric value by calling `use_facet < num_get < Elem, InIt>(getloc). get(InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator < Elem, Tr>`, and `val` has type **long**, **unsigned long**, or **void *** as needed.

If the converted value cannot be represented as the type of `val`, the function calls `setstate(failbit)`. In any case, the function returns ***this**.

The functions:

```
basic_istream& operator>>(float& val);
basic_istream& operator>>(double& val);
basic_istream& operator>>(long double& val);
```

each extract a field and convert it to a numeric value by calling `use_facet < num_get < Elem, InIt>(getloc). get(InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator < Elem, Tr>`, and `val` has type **double** or **long double** as needed.

If the converted value cannot be represented as the type of `val`, the function calls `setstate(failbit)`. In any case, it returns ***this**.

Example

```

// istream_basic_istream_op_is.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

ios_base& hex2( ios_base& ib )
{
    ib.unsetf( ios_base::dec );
    ib.setf( ios_base::hex );
    return ib;
}

basic_istream<char, char_traits<char> >& somefunc(basic_istream<char, char_traits<char> > &i)
{
    if ( i == cin )
    {
        cerr << "i is cin" << endl;
    }
    return i;
}

int main( )
{
    int i = 0;
    cin >> somefunc;
    cin >> i;
    cout << i << endl;
    cin >> hex2;
    cin >> i;
    cout << i << endl;
}

```

basic_istream::operator=

Assigns the `basic_istream` on the right side of the operator to this object. This is a move assignment involving an `rvalue` reference that does not leave a copy behind.

```
basic_istream& operator=(basic_istream&& right);
```

Parameters

right

An `rvalue` reference to a `basic_ifstream` object.

Return Value

Returns `*this`.

Remarks

The member operator calls `swap (right)` .

basic_istream::peek

Returns the next character to be read.

```
int_type peek();
```

Return Value

The next character that will be read.

Remarks

The unformatted input function extracts an element, if possible, as if by returning `rdbuf` -> `sgetc`. Otherwise, it returns `traits_type::eof`.

Example

```
// basic_istream_peek.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char c[10], c2;
    cout << "Type 'abcde': ";

    c2 = cin.peek( );
    cin.getline( &c[0], 9 );

    cout << c2 << " " << c << endl;
}
```

abcde

Type 'abcde': abcde
a abcde

basic_istream::putback

Puts a specified character into the stream.

```
basic_istream<Elem, Tr>& putback(
    char_type Ch);
```

Parameters

Ch

A character to put back into the stream.

Return Value

The stream (***this**).

Remarks

The [unformatted input function](#) puts back *Ch*, if possible, as if by calling `rdbuf` -> `sputbackc`. If `rdbuf` is a null pointer, or if the call to `sputbackc` returns `traits_type::eof`, the function calls `setstate(badbit)`. In any case, it returns ***this**.

Example

```
// basic_istream_putback.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char c[10], c2, c3;

    c2 = cin.get( );
    c3 = cin.get( );
    cin.putback( c2 );
    cin.getline( &c[0], 9 );
    cout << c << endl;
}
```

qwq

basic_istream::read

Reads a specified number of characters from the stream and stores them in an array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
basic_istream<Elem, Tr>& read(
    char_type* str,
    streamsize count);
```

Parameters

str

The array in which to read the characters.

count

The number of characters to read.

Return Value

The stream (`*this`).

Remarks

The unformatted input function extracts up to *count* elements and stores them in the array beginning at `_str`. Extraction stops early on end of file, in which case the function calls `setstate(failbit)`. In any case, it returns `*this`.

Example

```
// basic_istream_read.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
    char c[10];
    int count = 5;

    cout << "Type 'abcde': ";

    // Note: cin::read is potentially unsafe, consider
    // using cin::_Read_s instead.
    cin.read(&c[0], count);
    c[count] = 0;

    cout << c << endl;
}
```

```
abcde
```

```
Type 'abcde': abcde
abcde
```

basic_istream::readsome

Reads the specified number of character values.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
streamsize readsome(
    char_type* str,
    streamsize count);
```

Parameters

str

The array in which `readsome` stores the characters it reads.

count

The number of characters to read.

Return Value

The number of characters actually read, [gcount](#).

Remarks

This unformatted input function extracts up to *count* elements from the input stream and stores them in the array *str*.

This function does not wait for input. It reads whatever data is available.

Example


```

// basic_istream_readsome.cpp
// compile with: /EHsc /W3
#include <iostream>
using namespace std;

int main( )
{
    char c[10];
    int count = 5;

    cout << "Type 'abcdefgh': ";

    // cin.read blocks until user types input.
    // Note: cin::read is potentially unsafe, consider
    // using cin::_Read_s instead.
    cin.read(&c[0], 2);

    // Note: cin::readsome is potentially unsafe, consider
    // using cin::_Readsome_s instead.
    int n = cin.readsome(&c[0], count); // C4996
    c[n] = 0;
    cout << n << " characters read" << endl;
    cout << c << endl;
}

```

basic_istream::seekg

Moves the read position in a stream.

```

basic_istream<Elem, Tr>& seekg(pos_type pos);

basic_istream<Elem, Tr>& seekg(off_type off, ios_base::seekdir way);

```

Parameters

pos

The absolute position in which to move the read pointer.

off

An offset to move the read pointer relative to *way*.

way

One of the [ios_base::seekdir](#) enumerations.

Return Value

The stream (***this**).

Remarks

The first member function performs an absolute seek, the second member function performs a relative seek.

NOTE

Do not use the second member function with text files, because Standard C++ does not support relative seeks in text files.

If **fail** is false, the first member function calls **newpos** = **rdbuf** -> **pubseekpos**(**pos**), for some **pos_type** temporary object **newpos**. If **fail** is false, the second function calls **newpos** = **rdbuf** -> **pubseekoff**(**off**, **way**). In either case, if (**off_type**) **newpos** == (**off_type**)(-1) (the positioning operation fails), the function calls

`istr`. `setstate(failbit)`. Both functions return ***this**.

If `fail` is true, the member functions do nothing.

Example

```
// basic_istream_seekg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    ifstream file;
    char c, c1;

    file.open( "basic_istream_seekg.txt" );
    file.seekg(2); // seek to position 2
    file >> c;
    cout << c << endl;
}
```

basic_istream::sentry

The nested class describes an object whose declaration structures the formatted and unformatted input functions.

```
class sentry { public: explicit sentry( basic_istream<Elem, Tr>& _Istr, bool _Noskip = false); operator bool() const; };
```

Remarks

If `_Istr`. `good` is true, the constructor:

- Calls `_Istr`. `tie` -> `flush` if `_Istr`. `tie` is not a null pointer
- Effectively calls `ws(_Istr)` if `_Istr`. `flags&skipws` is nonzero

If, after any such preparation, `_Istr`. `good` is false, the constructor calls `_Istr`. `setstate(failbit)`. In any case, the constructor stores the value returned by `_Istr`. `good` in `status`. A later call to `operator bool` delivers this stored value.

basic_istream::swap

Exchanges the contents of two `basic_istream` objects.

```
void swap(basic_istream& right);
```

Parameters

right

An lvalue reference to a `basic_istream` object.

Remarks

The member function calls `basic_ios::swap(right)`. It also exchanges the extraction count with the extraction count for *right*.

basic_istream::sync

Synchronizes the input device associated with the stream with the stream's buffer.

```
int sync();
```

Return Value

If `rdbuf` is a null pointer, the function returns -1. Otherwise, it calls `rdbuf` -> `pubsync`. If that returns -1, the function calls `setstate(badbit)` and returns -1. Otherwise, the function returns zero.

basic_istream::tellg

Reports the current read position in the stream.

```
pos_type tellg();
```

Return Value

The current position in the stream.

Remarks

If `fail` is false, the member function returns `rdbuf` -> `pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type` (-1).

Example

```
// basic_istream_tellg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;
    ifstream file;
    char c;
    streamoff i;

    file.open("basic_istream_tellg.txt");
    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;

    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;
}
```

basic_istream::unget

Puts the most recently read character back into the stream.

```
basic_istream<Elem, Tr>& unget();
```

Return Value

The stream (***this**).

Remarks

The [unformatted input function](#) puts back the previous element in the stream, if possible, as if by calling `rdbuf`

-> [sungetc](#). If [rdbuf](#) is a null pointer, or if the call to `sungetc` returns `traits_type::eof`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

For information on how `ungetc` might fail, see [basic_streambuf::sungetc](#).

Example

```
// basic_istream_ungetc.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char c[10], c2;

    cout << "Type 'abc': ";
    c2 = cin.get( );
    cin.ungetc( );
    cin.getline( &c[0], 9 );
    cout << c << endl;
}
```

abc

Type 'abc': abc
abc

See also

[Thread Safety in the C++ Standard Library](#)

[istream Programming](#)

[istreams Conventions](#)

<iterator>

11/9/2018 • 6 minutes to read • [Edit Online](#)

Defines the iterator primitives, predefined iterators and stream iterators, as well as several supporting templates. The predefined iterators include insert and reverse adaptors. There are three classes of insert iterator adaptors: front, back, and general. They provide insert semantics rather than the overwrite semantics that the container member function iterators provide.

Syntax

```
#include <iterator>
```

Remarks

Iterators are a generalization of pointers, abstracting from their requirements in a way that allows a C++ program to work with different data structures in a uniform manner. Iterators act as intermediaries between containers and generic algorithms. Instead of operating on specific data types, algorithms are defined to operate on a range specified by a type of iterator. Any data structure that satisfies the requirements of the iterator may then be operated on by the algorithm. There are five types or categories of iterator, each with its own set of requirements and resulting functionality:

- Output: forward moving, may store but not retrieve values, provided by ostream and inserter.
- Input: forward moving, may retrieve but not store values, provided by istream.
- Forward: forward moving, may store and retrieve values.
- Bidirectional: forward and backward moving, may store and retrieve values, provided by list, set, multiset, map, and multimap.
- Random access: elements accessed in any order, may store and retrieve values, provided by vector, deque, string, and array.

Iterators that have greater requirements and so more powerful access to elements may be used in place of iterators with fewer requirements. For example, if a forward iterator is called for, then a random-access iterator may be used instead.

Visual Studio has added extensions to C++ Standard Library iterators to support a variety of debug mode situations for checked and unchecked iterators. For more information, see [Safe Libraries: C++ Standard Library](#).

Functions

FUNCTION	DESCRIPTION
advance	Increments an iterator by a specified number of positions.
back_inserter	Creates an iterator that can insert elements at the back of a specified container.
begin	Retrieves an iterator to the first element in a specified container.

FUNCTION	DESCRIPTION
<code>cbegin</code>	Retrieves a constant iterator to the first element in a specified container.
<code>cend</code>	Retrieves a constant iterator to the element that follows the last element in the specified container.
<code>distance</code>	Determines the number of increments between the positions addressed by two iterators.
<code>end</code>	Retrieves an iterator to the element that follows the last element in the specified container.
<code>front_inserter</code>	Creates an iterator that can insert elements at the front of a specified container.
<code>inserter</code>	An iterator adaptor that adds a new element to a container at a specified point of insertion.
<code>make_checked_array_iterator</code>	Creates a <code>checked_array_iterator</code> that can be used by other algorithms. Note: This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.
<code>make_move_iterator</code>	Returns a move iterator containing the provided iterator as its stored base iterator.
<code>make_unchecked_array_iterator</code>	Creates an <code>unchecked_array_iterator</code> that can be used by other algorithms. Note: This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.
<code>next</code>	Iterates a specified number of times and returns the new iterator position.
<code>prev</code>	Iterates in reverse a specified number of times and returns the new iterator position.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the iterator object on the left side of the operator is not equal to the iterator object on the right side.
<code>operator==</code>	Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.
<code>operator<</code>	Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.
<code>operator<=</code>	Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.

OPERATOR	DESCRIPTION
<code>operator></code>	Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.
<code>operator>=</code>	Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.
<code>operator+</code>	Adds an offset to an iterator and returns the new <code>reverse_iterator</code> addressing the inserted element at the new offset position.
<code>operator-</code>	Subtracts one iterator from another and returns the difference.

Classes

CLASS	DESCRIPTION
<code>back_insert_iterator</code>	The template class describes an output iterator object. It inserts elements into a container of type <code>Container</code> , which it accesses through the protected <code>pointer</code> object it stores called container.
<code>bidirectional_iterator_tag</code>	A class that provides a return type for an <code>iterator_category</code> function that represents a bidirectional iterator.
<code>checked_array_iterator</code>	A class that accesses an array using a random access, checked iterator. Note: This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.
<code>forward_iterator_tag</code>	A class that provides a return type for an <code>iterator_category</code> function that represents a forward iterator.
<code>front_insert_iterator</code>	The template class describes an output iterator object. It inserts elements into a container of type <code>Container</code> , which it accesses through the protected <code>pointer</code> object it stores called container.
<code>input_iterator_tag</code>	A class that provides a return type for an <code>iterator_category</code> function that represents an input iterator.
<code>insert_iterator</code>	The template class describes an output iterator object. It inserts elements into a container of type <code>Container</code> , which it accesses through the protected <code>pointer</code> object it stores called container. It also stores the protected <code>iterator</code> object, of class <code>Container::iterator</code> , called <code>iter</code> .

CLASS	DESCRIPTION
istream_iterator	The template class describes an input iterator object. It extracts objects of class <code>Ty</code> from an input stream, which it accesses through an object it stores, of type pointer to <code>basic_istream<Elem, Tr></code> .
istreambuf_iterator	The template class describes an input iterator object. It inserts elements of class <code>Elem</code> into an output stream buffer, which it accesses through an object it stores, of type pointer to <code>basic_streambuf<Elem, Tr></code> .
iterator	The template class is used as a base type for all iterators.
iterator_traits	A template helper class providing critical types that are associated with different iterator types so that they can be referred to in the same way.
move_iterator	A <code>move_iterator</code> object stores a random-access iterator of type <code>RandomIterator</code> . It behaves like a random-access iterator, except when dereferenced. The result of <code>operator*</code> is implicitly cast to <code>value_type&&</code> to make an rvalue reference.
ostream_iterator	The template class describes an output iterator object. It inserts objects of class <code>Type</code> into an output stream, which it accesses through an object it stores, of type pointer to <code>basic_ostream<Elem, Tr></code> .
ostreambuf_iterator Class	The template class describes an output iterator object. It inserts elements of class <code>Elem</code> into an output stream buffer, which it accesses through an object it stores, of type pointer to <code>basic_streambuf<Elem, Tr></code> .
output_iterator_tag	A class that provides a return type for <code>iterator_category</code> function that represents an output iterator.
random_access_iterator_tag	A class that provides a return type for <code>iterator_category</code> function that represents a random-access iterator.
reverse_iterator	The template class describes an object that behaves like a random-access iterator, only in reverse.
unchecked_array_iterator	A class that accesses an array using a random access, unchecked iterator. Note: This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<iterator> functions

11/9/2018 • 13 minutes to read • [Edit Online](#)

advance	back_inserter	begin
cbegin	cend	distance
end	front_inserter	inserter
make_checked_array_iterator	make_move_iterator	make_unchecked_array_iterator
next	prev	

advance

Increments an iterator by a specified number of positions.

```
template <class InputIterator, class Distance>
void advance(
    InputIterator& InIt,
    Distance Off);
```

Parameters

InIt

The iterator that is to be incremented and that must satisfy the requirements for an input iterator.

Off

An integral type that is convertible to the iterator's difference type and that specifies the number of increments the position of the iterator is to be advanced.

Remarks

The range advanced through must be nonsingular, where the iterators must be dereferenceable or past the end.

If the `InputIterator` satisfies the requirements for a bidirectional iterator type, then *Off* may be negative. If

`InputIterator` is an input or forward iterator type, *Off* must be nonnegative.

The advance function has constant complexity when `InputIterator` satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

```

// iterator_advance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    list<int> L;
    for ( i = 1 ; i < 9 ; ++i )
    {
        L.push_back ( i );
    }
    list<int>::iterator L_Iter, LPOS = L.begin ( );

    cout << "The list L is: ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;

    cout << "The iterator LPOS initially points to the first element: "
        << *LPOS << "." << endl;

    advance ( LPOS , 4 );
    cout << "LPOS is advanced 4 steps forward to point"
        << " to the fifth element: "
        << *LPOS << "." << endl;

    advance ( LPOS , -3 );
    cout << "LPOS is moved 3 steps back to point to the "
        << "2nd element: " << *LPOS << "." << endl;
}

```

```

The list L is: ( 1 2 3 4 5 6 7 8 ).
The iterator LPOS initially points to the first element: 1.
LPOS is advanced 4 steps forward to point to the fifth element: 5.
LPOS is moved 3 steps back to point to the 2nd element: 2.

```

back_inserter

Creates an iterator that can insert elements at the back of a specified container.

```

template <class Container>
back_insert_iterator<Container> back_inserter(Container& _Cont);

```

Parameters

_Cont

The container into which the back insertion is to be executed.

Return Value

A `back_insert_iterator` associated with the container object *_Cont*.

Remarks

Within the C++ Standard Library, the argument must refer to one of the three sequence containers that have the member function `push_back`: [deque Class](#), [list Class](#), or [vector Class](#).

Example

```
// iterator_back_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 3 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    // Insertions can be done with template function
    back_insert_iterator<vector<int> > backiter ( vec );
    *backiter = 30;
    backiter++;
    *backiter = 40;

    // Alternatively, insertions can be done with the
    // back_inserter_iterator member function
    back_inserter ( vec ) = 500;
    back_inserter ( vec ) = 600;

    cout << "After the insertions, the vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}
```

The initial vector vec is: (0 1 2).
After the insertions, the vector vec is: (0 1 2 30 40 500 600).

begin

Retrieves an iterator to the first element in a specified container.

```
template <class Container>
auto begin(Container& cont) \
    -> decltype(cont.begin());

template <class Container>
auto begin(const Container& cont) \
    -> decltype(cont.begin());

template <class Ty, class Size>
Ty *begin(Ty (& array)[Size]);
```

Parameters

cont

A container.

array

An array of objects of type `Ty`.

Return Value

The first two template functions return `cont.begin()`. The first function is non-constant; the second one is constant.

The third template function returns *array*.

Example

We recommend that you use this template function in place of container member `begin()` when more generic behavior is required.

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

template <typename C> void reverse_sort(C& c) {
    using std::begin;
    using std::end;

    std::sort(begin(c), end(c), std::greater<>());
}

template <typename C> void print(const C& c) {
    for (const auto& e : c) {
        std::cout << e << " ";
    }

    std::cout << "\n";
}

int main() {
    std::vector<int> v = { 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 };

    print(v);
    reverse_sort(v);
    print(v);

    std::cout << "--\n";

    int arr[] = { 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1 };

    print(arr);
    reverse_sort(arr);
    print(arr);
}
```

```
11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
52 40 34 26 20 17 16 13 11 10 8 5 4 2 1
--
23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
160 106 80 70 53 40 35 23 20 16 10 8 5 4 2 1
```

The function `reverse_sort` supports containers of any kind, in addition to regular arrays, because it calls the non-member version of `begin()`. If `reverse_sort` were coded to use the container member `begin()`:

```
template <typename C>
void reverse_sort(C& c) {
    using std::begin;
    using std::end;

    std::sort(c.begin(), c.end(), std::greater<>());
}
```

Then sending an array to it would cause this compiler error:

```
error C2228: left of '.begin' must have class/struct/union
```

cbegin

Retrieves a const iterator to the first element in a specified container.

```
template <class Container>
auto cbegin(const Container& cont)
    -> decltype(cont.begin());
```

Parameters

cont

A container or `initializer_list`.

Return Value

A constant `cont.begin()` .

Remarks

This function works with all C++ Standard Library containers and with [initializer_list](#).

You can use this member function in place of the `begin()` template function to guarantee that the return value is `const_iterator` . Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container or `initializer_list` of any kind that supports `begin()` and `cbegin()` .

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Retrieves a const iterator to the element that follows the last element in the specified container.

```
template <class Container>
auto cend(const Container& cont)
    -> decltype(cont.end());
```

Parameters

cont

A container or `initializer_list`.

Return Value

A constant `cont.end()` .

Remarks

This function works with all C++ Standard Library containers and with [initializer_list](#).

You can use this member function in place of the [end\(\)](#) template function to guarantee that the return value is `const_iterator` . Typically, it's used in conjunction with the [auto](#) type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container or `initializer_list` of any kind that supports `end()` and `cend()` .

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

distance

Determines the number of increments between the positions addressed by two iterators.

```
template <class InputIterator>
typename iterator_traits<InputIterator>::difference_type distance(InputIterator first, InputIterator last);
```

Parameters

first

The first iterator whose distance from the second is to be determined.

last

The second iterator whose distance from the first is to be determined.

Return Value

The number of times that *first* must be incremented until it equal *last*.

Remarks

The distance function has constant complexity when `InputIterator` satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

```

// iterator_distance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    list<int> L;
    for ( i = -1 ; i < 9 ; ++i )
    {
        L.push_back ( 2 * i );
    }
    list<int>::iterator L_Iter, LPOS = L.begin ( );

    cout << "The list L is: ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ) ; L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;

    cout << "The iterator LPOS initially points to the first element: "
        << *LPOS << "." << endl;

    advance ( LPOS , 7 );
    cout << "LPOS is advanced 7 steps forward to point "
        << " to the eighth element: "
        << *LPOS << "." << endl;

    list<int>::difference_type Ldiff ;
    Ldiff = distance ( L.begin ( ) , LPOS );
    cout << "The distance from L.begin( ) to LPOS is: "
        << Ldiff << "." << endl;
}

```

The list L is: (-2 0 2 4 6 8 10 12 14 16).
 The iterator LPOS initially points to the first element: -2.
 LPOS is advanced 7 steps forward to point to the eighth element: 12.
 The distance from L.begin() to LPOS is: 7.

end

Retrieves an iterator to the element that follows the last element in the specified container.

```

template <class Container>
auto end(Container& cont)
    -> decltype(cont.end());

template <class Container>
auto end(const Container& cont)
    -> decltype(cont.end());

template <class Ty, class Size>
Ty *end(Ty (& array)[Size]);

```

Parameters

cont

A container.

array

An array of objects of type `Ty`.

Return Value

The first two template functions return `cont.end()` (the first is non-constant and the second is constant).

The third template function returns `array + Size`.

Remarks

For a code example, see [begin](#).

front_inserter

Creates an iterator that can insert elements at the front of a specified container.

```
template <class Container>
front_insert_iterator<Container> front_inserter(Container& _Cont);
```

Parameters

_Cont

The container object whose front is having an element inserted.

Return Value

A `front_insert_iterator` associated with the container object *_Cont*.

Remarks

The member function [front_insert_iterator](#) of the `front_insert_iterator` class may also be used.

Within the C++ Standard Library, the argument must refer to one of the two sequence containers that have the member function `push_back`: [deque Class](#) or "list Class".

Example


```

// iterator_front_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for ( i = -1 ; i < 9 ; ++i )
    {
        L.push_back ( i );
    }

    cout << "The list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;

    // Using the template function to insert an element
    front_insert_iterator< list< int> > Iter(L);
    *Iter = 100;

    // Alternatively, you may use the front_insert member function
    front_inserter ( L ) = 200;

    cout << "After the front insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;
}

```

```

The list L is:
( -1 0 1 2 3 4 5 6 7 8 ).
After the front insertions, the list L is:
( 200 100 -1 0 1 2 3 4 5 6 7 8 ).

```

inserter

A helper template function that lets you use `inserter(_Cont, _Where)` instead of

`insert_iterator<Container>(_Cont, _Where)`.

```

template <class Container>
insert_iterator<Container>
inserter(
    Container& _Cont,
    typename Container::iterator _Where);

```

Parameters

_Cont

The container to which new elements are to be added.

_Where

An iterator locating the point of insertion.

Remarks

The template function returns `insert_iterator<Container>(_Cont, _Where)` .

Example

```
// iterator_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for (i = 2 ; i < 5 ; ++i )
    {
        L.push_back ( 10 * i );
    }

    cout << "The list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ) ; L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;

    // Using the template version to insert an element
    insert_iterator<list<int> > Iter( L, L.begin ( ) );
    *Iter = 1;

    // Alternatively, using the member function to insert an element
    inserter ( L, L.end ( ) ) = 500;

    cout << "After the insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ) ; L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
```

```
The list L is:
( 20 30 40 ).
After the insertions, the list L is:
( 1 20 30 40 500 ).
```

make_checked_array_iterator

Creates a `checked_array_iterator` that can be used by other algorithms.

NOTE

This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

```
template <class Iter>
checked_array_iterator<Iter>
    make_checked_array_iterator(
Iter Ptr,
    size_t Size,
    size_t Index = 0);
```

Parameters

Ptr

A pointer to the destination array.

Size

The size of the destination array.

Index

Optional index into the array.

Return Value

An instance of `checked_array_iterator`.

Remarks

The `make_checked_array_iterator` function is defined in the `stdext` namespace.

This function takes a raw pointer—which would ordinarily cause concern about bounds overrun—and wraps it in a [checked_array_iterator](#) class that does checking. Because that class is marked as checked, the C++ Standard Library doesn't warn about it. For more information and code examples, see [Checked Iterators](#).

Example

In the following example, a [vector](#) is created and populated with 10 items. The contents of the vector are copied into an array by using the copy algorithm, and then `make_checked_array_iterator` is used to specify the destination. This is followed by an intentional violation of the bounds checking so that a debug assertion failure is triggered.

```

// make_checked_array_iterator.cpp
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iterator> // stdext::make_checked_array_iterator
#include <memory> // std::make_unique
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    const size_t dest_size = 10;
    // Old-school but not exception safe, favor make_unique<int[]>
    // int* dest = new int[dest_size];
    unique_ptr<int[]> updest = make_unique<int[]>(dest_size);
    int* dest = updest.get(); // get a raw pointer for the demo

    vector<int> v;

    for (int i = 0; i < dest_size; ++i) {
        v.push_back(i);
    }
    print("vector v: ", v);

    copy(v.begin(), v.end(), stdext::make_checked_array_iterator(dest, dest_size));

    cout << "int array dest: ";
    for (int i = 0; i < dest_size; ++i) {
        cout << dest[i] << " ";
    }
    cout << endl;

    // Add another element to the vector to force an overrun.
    v.push_back(10);
    // The next line causes a debug assertion when it executes.
    copy(v.begin(), v.end(), stdext::make_checked_array_iterator(dest, dest_size));
}

```

make_move_iterator

Creates a `move iterator` that contains the provided iterator as the `stored` iterator.

```

template <class Iterator>
move_iterator<Iterator>
make_move_iterator(const Iterator& _It);

```

Parameters

`_It`

The iterator stored in the new move iterator.

Remarks

The template function returns `move_iterator` `<Iterator>(_It)`.

make_unchecked_array_iterator

Creates an [unchecked_array_iterator](#) that can be used by other algorithms.

NOTE

This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

```
template <class Iter>
unchecked_array_iterator<Iter>
    make_unchecked_array_iterator(Iter Ptr);
```

Parameters

Ptr

A pointer to the destination array.

Return Value

An instance of `unchecked_array_iterator`.

Remarks

The `make_unchecked_array_iterator` function is defined in the `stdext` namespace.

This function takes a raw pointer and wraps it in a class that performs no checking and therefore optimizes away to nothing, but it also silences compiler warnings such as [C4996](#). Therefore, this is a targeted way to deal with unchecked-pointer warnings without globally silencing them or incurring the cost of checking. For more information and code examples, see [Checked Iterators](#).

Example

In the following example, a [vector](#) is created and populated with 10 items. The contents of the vector are copied into an array by using the copy algorithm, and then `make_unchecked_array_iterator` is used to specify the destination.

```

// make_unchecked_array_iterator.cpp
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iterator> // stdext::make_unchecked_array_iterator
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    const size_t dest_size = 10;
    int *dest = new int[dest_size];
    vector<int> v;

    for (int i = 0; i < dest_size; ++i) {
        v.push_back(i);
    }
    print("vector v: ", v);

    // COMPILER WARNING SILENCED: stdext::unchecked_array_iterator is marked as checked in debug mode
    // (it performs no checking, so an overrun will trigger undefined behavior)
    copy(v.begin(), v.end(), stdext::make_unchecked_array_iterator(dest));

    cout << "int array dest: ";
    for (int i = 0; i < dest_size; ++i) {
        cout << dest[i] << " ";
    }
    cout << endl;

    delete[] dest;
}

```

next

Iterates a specified number of times and returns the new iterator position.

```

template <class InputIterator>
InputIterator next(
    InputIterator first,
    typename iterator_traits<InputIterator>::difference_type _Off = 1);

```

Parameters

first

The current position.

_Off

The number of times to iterate.

Return Value

Returns the new iterator position after iterating *_Off* times.

Remarks

The template function returns `next` incremented *_Off* times

prev

Iterates in reverse a specified number of times and returns the new iterator position.

```
template <class BidirectionalIterator>
BidirectionalIterator prev(
    BidirectionalIterator first,
    typename iterator_traits<BidirectionalIterator>::difference_type _Off = 1);
```

Parameters

first

The current position.

_Off

The number of times to iterate.

Remarks

The template function returns `next` decremented `off` times.

See also

[<iterator>](#)

<iterator> operators

10/31/2018 • 15 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator+</code>
<code>operator-</code>	<code>operator==</code>	

operator!=

Tests if the iterator object on the left side of the operator is not equal to the iterator object on the right side.

```
template <class RandomIterator>
bool operator!=(const reverse_iterator<RandomIterator>& left, const reverse_iterator<RandomIterator>& right);

template <class Type, class CharType, class Traits, class Distance>
bool operator!=(const istream_iterator<Type, CharType, Traits, Distance>& left, const istream_iterator<Type,
CharType, Traits, Distance>& right);

template <class CharType, class Tr>
bool operator!=(const istreambuf_iterator<CharType, Traits>& left, const istreambuf_iterator<CharType, Traits>&
right);
```

Parameters

left

An object of type `iterator`.

right

An object of type `iterator`.

Return Value

true if the iterator objects are not equal; **false** if the iterator objects are equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

Example


```

// iterator_op_ne.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 9 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector<int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 != rVPOS2 )
        cout << "The iterators are not equal." << endl;
    else
        cout << "The iterators are equal." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 != rVPOS2 )
        cout << "The iterators are not equal." << endl;
    else
        cout << "The iterators are equal." << endl;
}

```

```

The vector vec is: ( 1 2 3 4 5 6 7 8 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 8.
The iterators are equal.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 7.
The iterators are not equal.

```

operator==

Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.

```

template <class RandomIterator1, class RandomIterator2>
bool operator==(
    const move_iterator<RandomIterator1>& left,
    const move_iterator<RandomIterator2>& right);

template <class RandomIterator1, class RandomIterator2>
bool operator==(
    const reverse_iterator<RandomIterator1>& left,
    const reverse_iterator<RandomIterator2>& right);

template <class Type, class CharType, class Traits, class Distance>
bool operator==(
    const istream_iterator<Type, CharType, Traits, Distance>& left,
    const istream_iterator<Type, CharType, Traits, Distance>& right);

template <class CharType, class Tr>
bool operator==(
    const istreambuf_iterator<CharType, Traits>& left,
    const istreambuf_iterator<CharType, Traits>& right);

```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator objects are equal; **false** if the iterator objects are not equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

The first two template operators return true only if both *left* and *right* store the same iterator. The third template operator returns true only if both *left* and *right* store the same stream pointer. The fourth template operator returns

`left.equal (right) .`

Example

```

// iterator_op_eq.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector<int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector<int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 == rVPOS2 )
        cout << "The iterators are equal." << endl;
    else
        cout << "The iterators are not equal." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 == rVPOS2 )
        cout << "The iterators are equal." << endl;
    else
        cout << "The iterators are not equal." << endl;
}

```

```

The vector vec is: ( 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
The iterators are equal.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 8.
The iterators are not equal.

```

operator<

Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.

```

template <class RandomIterator>
bool operator<(const reverse_iterator<RandomIterator>& left, const reverse_iterator<RandomIterator>& right);

```

Parameters

left

An object of type `iterator` .

right

An object of type `iterator` .

Return Value

true if the iterator on the left side of the expression is less than the iterator on the right side of the expression; **false** if it is greater than or equal to the iterator on the right.

Remarks

One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object is not less than another if it addresses either the same element as the other iterator object or an element that occurs later in the container than the element addressed by the other iterator object.

Example

```

// iterator_op_lt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector<int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector<int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( );

    cout << "The iterators rVPOS1& rVPOS2 initially point to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 < rVPOS2 )
        cout << "The iterator rVPOS1 is less than"
            << " the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is not less than"
            << " the iterator rVPOS2." << endl;

    rVPOS2++;
    cout << "The iterator rVPOS2 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if ( rVPOS1 < rVPOS2 )
        cout << "The iterator rVPOS1 is less than"
            << " the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is not less than"
            << " the iterator rVPOS2." << endl;
}

```

The initial vector vec is: (0 2 4 6 8 10).
 The iterators rVPOS1& rVPOS2 initially point to the first element
 in the reversed sequence: 10.
 The iterator rVPOS1 is not less than the iterator rVPOS2.
 The iterator rVPOS2 now points to the second element
 in the reversed sequence: 8.
 The iterator rVPOS1 is less than the iterator rVPOS2.

operator<=

Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.

```
template <class RandomIterator>
bool operator<=(const reverse_iterator<RandomIterator>& left, const reverse_iterator<RandomIterator>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is less than or equal to the iterator on the right side of the expression; **false** if it is greater than the iterator on the right.

Remarks

One iterator object is less than or equal to another if it addresses the same element or an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object.

Example

```

// iterator_op_le.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 6 ; ++i ) {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ) + 1,
        rVPOS2 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the "
        << "second element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    cout << "The iterator rVPOS2 initially points to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if ( rVPOS1 <= rVPOS2 )
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;

    rVPOS2++;
    cout << "The iterator rVPOS2 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if ( rVPOS1 <= rVPOS2 )
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
}

```

The initial vector vec is: (0 2 4 6 8 10).
 The iterator rVPOS1 initially points to the second element
 in the reversed sequence: 8.
 The iterator rVPOS2 initially points to the first element
 in the reversed sequence: 10.
 The iterator rVPOS1 is greater than the iterator rVPOS2.
 The iterator rVPOS2 now points to the second element
 in the reversed sequence: 8.
 The iterator rVPOS1 is less than or equal to the iterator rVPOS2.

operator>

Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.

```
template <class RandomIterator>
bool operator>(const reverse_iterator<RandomIterator>& left, const reverse_iterator<RandomIterator>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is greater than the iterator on the right side of the expression;

false if it is less than or equal to the iterator on the right.

Remarks

One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object. One iterator object is not greater than another if it addresses either the same element as the other iterator object or an element that occurs earlier in the container than the element addressed by the other iterator object.

Example


```

// iterator_op_gt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 6 ; ++i ) {
        vec.push_back ( 2 * i );
    }

    vector<int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector<int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( );

    cout << "The iterators rVPOS1 & rVPOS2 initially point to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 > rVPOS2 )
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 > rVPOS2 )
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
}

```

```

The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1 & rVPOS2 initially point to the first element
in the reversed sequence: 10.
The iterator rVPOS1 is less than or equal to the iterator rVPOS2.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is greater than the iterator rVPOS2.

```

operator>=

Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.

```
template <class RandomIterator>
bool operator>=(const reverse_iterator<RandomIterator>& left, const reverse_iterator<RandomIterator>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is greater than or equal to the iterator on the right side of the expression; **false** if it is less than the iterator on the right.

Remarks

One iterator object is greater than or equal to another if it addresses the same element or an element that occurs later in the container than the element addressed by the other iterator object. One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object.

Example

```

// iterator_op_ge.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 6 ; ++i ) {
        vec.push_back ( 2 * i );
    }

    vector<int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector<int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( ) + 1;

    cout << "The iterator rVPOS1 initially points to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    cout << "The iterator rVPOS2 initially points to the "
        << "second element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if ( rVPOS1 >= rVPOS2 )
        cout << "The iterator rVPOS1 is greater than or "
            << "equal to the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is less than "
            << "the iterator rVPOS2." << endl;

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if ( rVPOS1 >= rVPOS2 )
        cout << "The iterator rVPOS1 is greater than or "
            << "equal to the iterator rVPOS2." << endl;
    else
        cout << "The iterator rVPOS1 is less than "
            << "the iterator rVPOS2." << endl;
}

```

The initial vector vec is: (0 2 4 6 8 10).
 The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
 The iterator rVPOS2 initially points to the second element
 in the reversed sequence: 8.
 The iterator rVPOS1 is less than the iterator rVPOS2.
 The iterator rVPOS1 now points to the second element
 in the reversed sequence: 8.
 The iterator rVPOS1 is greater than or equal to the iterator rVPOS2.

operator+

Adds an offset to an iterator and returns a `move_iterator` or a `reverse_iterator` addressing the inserted element at the new offset position.

```
template <class RandomIterator, class Diff>
move_iterator<RandomIterator>
operator+(
    Diff _Off,
    const move_iterator<RandomIterator>& right);

template <class RandomIterator>
reverse_iterator<RandomIterator>
operator+(
    Diff _Off,
    const reverse_iterator<RandomIterator>& right);
```

Parameters

_Off

The number of positions the const `move_iterator` or const `reverse_iterator` is to be offset.

right

The iterator to be offset.

Return Value

Returns the sum *right* + *_Off*.

Example

```

// iterator_op_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 0 ; i < 6 ; ++i ) {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    vector<int>::difference_type diff = 4;
    rVPOS1 = diff +rVPOS1;

    cout << "The iterator rVPOS1 now points to the fifth "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}

```

```

The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
The iterator rVPOS1 now points to the fifth element
in the reversed sequence: 2.

```

operator-

Subtracts one iterator from another and returns the difference.

```

template <class RandomIterator1, class RandomIterator2>
Tdiff operator-(
    const move_iterator<RandomIterator1>& left,
    const move_iterator<RandomIterator2>& right);

template <class RandomIterator1, class RandomIterator2>
Tdiff operator-(
    const reverse_iterator<RandomIterator1>& left,
    const reverse_iterator<RandomIterator2>& right);

```

Parameters

left

An iterator.

right

An iterator.

Return Value

The difference between two iterators .

Remarks

The first template operator returns `left.base() - right.base()` .

The second template operator returns `right.current - left.current` .

`Tdiff` is determined by the type of the returned expression. Otherwise, it is `RandomIterator1::difference_type` .

Example

```
// iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 0 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;

    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( ),
        rVPOS2 = vec.rbegin ( );

    cout << "The iterators rVPOS1 & rVPOS2 initially point to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    for (i = 1; i < 5; ++i)
    {
        rVPOS2++;
    }
    cout << "The iterator rVPOS2 now points to the fifth "
        << "element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    vector<int>::difference_type diff = rVPOS2 - rVPOS1;
    cout << "The difference: rVPOS2 - rVPOS1= "
        << diff << "." << endl;
}
```

The initial vector vec is: (0 2 4 6 8 10).
The iterators rVPOS1 & rVPOS2 initially point to the first element
in the reversed sequence: 10.
The iterator rVPOS2 now points to the fifth element
in the reversed sequence: 2.
The difference: rVPOS2 - rVPOS1= 4.

See also

[<iterator>](#)

back_insert_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the back end of a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The `back_insert_iterator` class is templated on the type of container.

Syntax

```
template <class Container>
class back_insert_iterator;
```

Parameters

Container

The type of container into the back of which elements are to be inserted by a `back_insert_iterator`.

Remarks

The container must satisfy the requirements for a back insertion sequence where it is possible to insert elements at the end of the sequence in amortized constant time. C++ Standard Library sequence containers defined by the [deque Class](#), [list Class](#) and [vector Class](#) provide the needed `push_back` member function and satisfy these requirements. These three containers as well as strings may each be adapted to use with `back_insert_iterator`s. A `back_insert_iterator` must always be initialized with its container.

Constructors

CONSTRUCTOR	DESCRIPTION
back_insert_iterator	Constructs a <code>back_insert_iterator</code> that inserts elements after the last element in a container.

Typedefs

TYPE NAME	DESCRIPTION
container_type	A type that provides a container for the <code>back_insert_iterator</code> .
reference	A type that provides a reference for the <code>back_insert_iterator</code> .

Operators

OPERATOR	DESCRIPTION
operator*	Dereferencing operator used to implement the output iterator expression <code>*i = x</code> for a back insertion.

OPERATOR	DESCRIPTION
<code>operator++</code>	Increments the <code>back_insert_iterator</code> to the next location into which a value may be stored.
<code>operator=</code>	Assignment operator used to implement the output iterator expression <code>* i = x</code> for a back insertion.

Requirements

Header: `<iterator>`

Namespace: `std`

`back_insert_iterator::back_insert_iterator`

Constructs a `back_insert_iterator` that inserts elements after the last element in a container.

```
explicit back_insert_iterator(Container& _Cont);
```

Parameters

_Cont

The container that the `back_insert_iterator` is to insert an element into.

Return Value

A `back_insert_iterator` for the parameter container.

Example

```

// back_insert_iterator_back_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 4 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The initial vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    // Insertions with member function
    back_inserter ( vec ) = 40;
    back_inserter ( vec ) = 50;

    // Alternatively, insertions can be done with template function
    back_insert_iterator<vector<int> > backiter ( vec );
    *backiter = 600;
    backiter++;
    *backiter = 700;

    cout << "After the insertions, the vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}

```

The initial vector vec is: (1 2 3).
 After the insertions, the vector vec is: (1 2 3 40 50 600 700).

back_insert_iterator::container_type

A type that provides a container for the `back_insert_iterator`.

```

typedef Container
container_type;

```

Remarks

The type is a synonym for the template parameter **Container**.

Example

```

// back_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 4 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The original vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    back_insert_iterator<vector<int> >::container_type> vec1 = vec;
    back_inserter ( vec1 ) = 40;

    cout << "After the insertion, the vector is: ( ";
    for ( vIter = vec1.begin ( ) ; vIter != vec1.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}

```

The original vector vec is: (1 2 3).
 After the insertion, the vector is: (1 2 3 40).

back_insert_iterator::operator*

Dereferencing operator used to implement the output iterator expression $*i = x$.

```
back_insert_iterator<Container>& operator*();
```

Return Value

A reference to the element inserted at the back of the container.

Remarks

Used to implement the output iterator expression ***Iter = value**. If **Iter** is an iterator that addresses an element in a sequence, then ***Iter = value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```

// back_insert_iterator_back_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 4 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    back_insert_iterator<vector<int> > backiter ( vec );
    *backiter = 10;
    backiter++;      // Increment to the next element
    *backiter = 20;
    backiter++;

    cout << "After the insertions, the vector vec becomes: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}

```

The vector vec is: (1 2 3).
 After the insertions, the vector vec becomes: (1 2 3 10 20).

back_insert_iterator::operator++

Increments the `back_insert_iterator` to the next location into which a value may be stored.

```

back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);

```

Return Value

A `back_insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```

// back_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 3 ; ++i )
    {
        vec.push_back ( 10 * i );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    back_insert_iterator<vector<int> > backiter ( vec );
    *backiter = 30;
    backiter++;      // Increment to the next element
    *backiter = 40;
    backiter++;

    cout << "After the insertions, the vector vec becomes: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}

```

The vector vec is: (10 20).
 After the insertions, the vector vec becomes: (10 20 30 40).

back_insert_iterator::operator=

Appends or pushes a value onto the back end of a container.

```

back_insert_iterator<Container>& operator=(typename Container::const_reference val);
back_insert_iterator<Container>& operator=(typename Container::value_type&& val);

```

Parameters

val

The value to be inserted into the container.

Return Value

A reference to the last element inserted at the back of the container.

Remarks

The first member operator evaluates `Container.push_back(val)`,

then returns `*this`. The second member operator evaluates

```

container->push_back((typename Container::value_type&&)val) ,

```

then returns `*this`.

Example

```
// back_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 4 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    back_insert_iterator<vector<int> > backiter ( vec );
    *backiter = 10;
    backiter++;      // Increment to the next element
    *backiter = 20;
    backiter++;

    cout << "After the insertions, the vector vec becomes: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}
```

back_insert_iterator::reference

A type that provides a reference for the `back_insert_iterator`.

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

```

// back_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 4 ; ++i )
    {
        vec.push_back ( i );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    back_insert_iterator<vector<int> >::reference
        RefLast = *(vec.end ( ) - 1 );
    cout << "The last element in the vector vec is: "
        << RefLast << "." << endl;
}

```

```

The vector vec is: ( 1 2 3 ).
The last element in the vector vec is: 3.

```

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

bidirectional_iterator_tag Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that provides a return type for `iterator_category` function that represents a bidirectional iterator.

Syntax

```
struct bidirectional_iterator_tag : public forward_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument, so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits < Iterator >::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as `iterator < Iter >::iterator_category` when `Iter` describes an object that can serve as a bidirectional iterator.

Example

See [random_access_iterator_tag](#) for an example of how to use `bidirectional_iterator_tag`.

Requirements

Header: `<iterator>`

Namespace: `std`

See also

[forward_iterator_tag](#) Struct

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

checked_array_iterator Class

3/28/2019 • 13 minutes to read • [Edit Online](#)

The `checked_array_iterator` class allows you to transform an array or pointer into a checked iterator. Use this class as a wrapper (using the [make_checked_array_iterator](#) function) for raw pointers or arrays as a targeted way to provide checking and to manage unchecked pointer warnings instead of globally silencing these warnings. If necessary, you can use the unchecked version of this class, [unchecked_array_iterator](#).

NOTE

This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension. For an example demonstrating how to write code that does not require the use of this class, see the second example below.

Syntax

```
template <class _Iterator>
class checked_array_iterator;
```

Remarks

This class is defined in the [stdext](#) namespace.

For more information and example code on the checked iterator feature, see [Checked Iterators](#).

Example

The following sample shows how to define and use a checked array iterator.

If the destination is not large enough to hold all the elements being copied, such as would be the case if you changed the line:

```
copy(a, a + 5, checked_array_iterator<int*>(b, 5));
```

to

```
copy(a, a + 5, checked_array_iterator<int*>(b, 4));
```

A runtime error will occur.

```

// compile with: /EHsc /W4 /MTd
#include <algorithm>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[]={0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b, 5));

    cout << "(";
    for (int i = 0 ; i < 5 ; i++)
        cout << " " << b[i];
    cout << " )" << endl;

    // constructor example
    checked_array_iterator<int*> checked_out_iter(b, 5);
    copy(a, a + 5, checked_out_iter);

    cout << "(";
    for (int i = 0 ; i < 5 ; i++)
        cout << " " << b[i];
    cout << " )" << endl;
}
/* Output:
( 0 1 2 3 4 )
( 0 1 2 3 4 )
*/

```

Example

To avoid the need for the `checked_array_iterator` class when using C++ Standard Library algorithms, consider using a `vector` instead of a dynamically allocated array. The following example demonstrates how to do this.

```
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    std::vector<int> v(10);
    int *arr = new int[10];
    for (int i = 0; i < 10; ++i)
    {
        v[i] = i;
        arr[i] = i;
    }

    // std::copy(v.begin(), v.end(), arr); will result in
    // warning C4996. To avoid this warning while using int *,
    // use the Microsoft extension checked_array_iterator.
    std::copy(v.begin(), v.end(),
              stdext::checked_array_iterator<int *>(arr, 10));

    // Instead of using stdext::checked_array_iterator and int *,
    // consider using std::vector to encapsulate the array. This will
    // result in no warnings, and the code will be portable.
    std::vector<int> arr2(10);    // Similar to int *arr = new int[10];
    std::copy(v.begin(), v.end(), arr2.begin());

    for (int j = 0; j < arr2.size(); ++j)
    {
        cout << " " << arr2[j];
    }
    cout << endl;

    return 0;
}
/* Output:
0 1 2 3 4 5 6 7 8 9
*/
```

Constructors

CONSTRUCTOR	DESCRIPTION
checked_array_iterator	Constructs a default <code>checked_array_iterator</code> or a <code>checked_array_iterator</code> from an underlying iterator.

Typedefs

TYPE NAME	DESCRIPTION
difference_type	A type that provides the difference between two <code>checked_array_iterator</code> s referring to elements within the same container.
pointer	A type that provides a pointer to an element addressed by a <code>checked_array_iterator</code> .

TYPE NAME	DESCRIPTION
<code>reference</code>	A type that provides a reference to an element addressed by a <code>checked_array_iterator</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>base</code>	Recovers the underlying iterator from its <code>checked_array_iterator</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator==</code>	Tests two <code>checked_array_iterator</code> s for equality.
<code>operator!=</code>	Tests two <code>checked_array_iterator</code> s for inequality.
<code>operator<</code>	Tests if the <code>checked_array_iterator</code> on the left side of the operator is less than the <code>checked_array_iterator</code> on the right side.
<code>operator></code>	Tests if the <code>checked_array_iterator</code> on the left side of the operator is greater than the <code>checked_array_iterator</code> on the right side.
<code>operator<=</code>	Tests if the <code>checked_array_iterator</code> on the left side of the operator is less than or equal to the <code>checked_array_iterator</code> on the right side.
<code>operator>=</code>	Tests if the <code>checked_array_iterator</code> on the left side of the operator is greater than or equal to the <code>checked_array_iterator</code> on the right side.
<code>operator*</code>	Returns the element that a <code>checked_array_iterator</code> addresses.
<code>operator-></code>	Returns a pointer to the element addressed by the <code>checked_array_iterator</code> .
<code>operator++</code>	Increments the <code>checked_array_iterator</code> to the next element.
<code>operator--</code>	Decrements the <code>checked_array_iterator</code> to the previous element.
<code>operator+=</code>	Adds a specified offset to a <code>checked_array_iterator</code> .
<code>operator+</code>	Adds an offset to an iterator and returns the new <code>checked_array_iterator</code> addressing the inserted element at the new offset position.

OPERATOR	DESCRIPTION
<code>operator-=</code>	Decrements a specified offset from a <code>checked_array_iterator</code> .
<code>operator-</code>	Decrements an offset from an iterator and returns the new <code>checked_array_iterator</code> addressing the inserted element at the new offset position.
<code>operator[]</code>	Returns a reference to an element offset from the element addressed by a <code>checked_array_iterator</code> by a specified number of positions.

Requirements

Header: `<iterator>`

Namespace: `stdext`

`checked_array_iterator::base`

Recovers the underlying iterator from its `checked_array_iterator`.

```
_Iterator base() const;
```

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_base.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main() {
    using namespace std;

    int V1[10];

    for (int i = 0; i < 10 ; i++)
        V1[i] = i;

    int* bpos;

    stdext::checked_array_iterator<int*> rpos(V1, 10);
    rpos++;

    bpos = rpos.base ( );
    cout << "The iterator underlying rpos is bpos & it points to: "
         << *bpos << "." << endl;
}
/* Output:
The iterator underlying rpos is bpos & it points to: 1.
*/
```

checked_array_iterator::checked_array_iterator

Constructs a default `checked_array_iterator` or a `checked_array_iterator` from an underlying iterator.

```
checked_array_iterator();

checked_array_iterator(
    IIterator ptr,
    size_t size,
    size_t index = 0);
```

Parameters

ptr

A pointer to the array.

size

The size of the array.

index

(Optional) An element in the array, to initialize the iterator. By default, the iterator is initialized to the first element in the array.

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_ctor.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    for (int i = 0 ; i < 5 ; i++)
        cout << b[i] << " ";
    cout << endl;

    checked_array_iterator<int*> checked_output_iterator(b,5);
    copy (a, a + 5, checked_output_iterator);
    for (int i = 0 ; i < 5 ; i++)
        cout << b[i] << " ";
    cout << endl;

    checked_array_iterator<int*> checked_output_iterator2(b,5,3);
    cout << *checked_output_iterator2 << endl;
}
/* Output:
0 1 2 3 4
0 1 2 3 4
3
*/
```

checked_array_iterator::difference_type

A type that provides the difference between two `checked_array_iterator`s referring to elements within the same container.

```
typedef typename iterator_traits<_Iterator>::difference_type difference_type;
```

Remarks

The `checked_array_iterator` difference type is the same as the iterator difference type.

See [checked_array_iterator::operator\[\]](#) for a code sample.

For more information, see [Checked Iterators](#).

checked_array_iterator::operator==

Tests two `checked_array_iterator`s for equality.

```
bool operator==(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` against which to check for equality.

Remarks

For more information, see [Checked Iterators](#).

Example

```

// checked_array_iterators_opeq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);
    checked_array_iterator<int*> checked_output_iterator2(b,5);

    if (checked_output_iterator2 == checked_output_iterator)
        cout << "checked_array_iterators are equal" << endl;
    else
        cout << "checked_array_iterators are not equal" << endl;

    copy (a, a + 5, checked_output_iterator);
    checked_output_iterator++;

    if (checked_output_iterator2 == checked_output_iterator)
        cout << "checked_array_iterators are equal" << endl;
    else
        cout << "checked_array_iterators are not equal" << endl;
}
/* Output:
checked_array_iterators are equal
checked_array_iterators are not equal
*/

```

checked_array_iterator::operator!=

Tests two `checked_array_iterator`s for inequality.

```
bool operator!=(const checked_array_iterator<Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` against which to check for inequality.

Remarks

For more information, see [Checked Iterators](#).

Example


```

// checked_array_iterators_opneq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);
    checked_array_iterator<int*> checked_output_iterator2(b,5);

    if (checked_output_iterator2 != checked_output_iterator)
        cout << "checked_array_iterators are not equal" << endl;
    else
        cout << "checked_array_iterators are equal" << endl;

    copy (a, a + 5, checked_output_iterator);
    checked_output_iterator++;

    if (checked_output_iterator2 != checked_output_iterator)
        cout << "checked_array_iterators are not equal" << endl;
    else
        cout << "checked_array_iterators are equal" << endl;
}
/* Output:
checked_array_iterators are equal
checked_array_iterators are not equal
*/

```

checked_array_iterator::operator<

Tests if the `checked_array_iterator` on the left side of the operator is less than the `checked_array_iterator` on the right side.

```
bool operator<(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` against which to check for inequality.

Remarks

For more information, see [Checked Iterators](#).

Example

```

// checked_array_iterators_oplt.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);
    checked_array_iterator<int*> checked_output_iterator2(b,5);

    if (checked_output_iterator2 < checked_output_iterator)
        cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
    else
        cout << "checked_output_iterator2 is not less than checked_output_iterator" << endl;

    copy (a, a + 5, checked_output_iterator);
    checked_output_iterator++;

    if (checked_output_iterator2 < checked_output_iterator)
        cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
    else
        cout << "checked_output_iterator2 is not less than checked_output_iterator" << endl;
}
/* Output:
checked_output_iterator2 is not less than checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
*/

```

checked_array_iterator::operator>

Tests if the `checked_array_iterator` on the left side of the operator is greater than the `checked_array_iterator` on the right side.

```
bool operator>(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` to compare against.

Remarks

See [checked_array_iterator::operator<](#) for a code sample.

For more information, see [Checked Iterators](#).

checked_array_iterator::operator<=

Tests if the `checked_array_iterator` on the left side of the operator is less than or equal to the `checked_array_iterator` on the right side.

```
bool operator<=(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` to compare against.

Remarks

See `checked_array_iterator::operator>=` for a code sample.

For more information, see [Checked Iterators](#).

checked_array_iterator::operator>=

Tests if the `checked_array_iterator` on the left side of the operator is greater than or equal to the `checked_array_iterator` on the right side.

```
bool operator>=(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` to compare against.

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_opgteq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);
    checked_array_iterator<int*> checked_output_iterator2(b,5);

    if (checked_output_iterator2 >= checked_output_iterator)
        cout << "checked_output_iterator2 is greater than or equal to checked_output_iterator" << endl;
    else
        cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;

    copy (a, a + 5, checked_output_iterator);
    checked_output_iterator++;

    if (checked_output_iterator2 >= checked_output_iterator)
        cout << "checked_output_iterator2 is greater than or equal to checked_output_iterator" << endl;
    else
        cout << "checked_output_iterator2 is less than checked_output_iterator" << endl;
}
/* Output:
checked_output_iterator2 is greater than or equal to checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
*/
```

checked_array_iterator::operator*

Returns the element that a `checked_array_iterator` addresses.

```
reference operator*() const;
```

Return Value

The value of the element addressed by the `checked_array_iterator`.

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
    int a[] = {0, 1, 2, 3, 4};
    int b[5];
    pair<int, int> c[1];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    for (int i = 0 ; i < 5 ; i++)
        cout << b[i] << endl;

    c[0].first = 10;
    c[0].second = 20;

    checked_array_iterator<int*> checked_output_iterator(b,5);
    checked_array_iterator<int*>::pointer p = &(*checked_output_iterator);
    checked_array_iterator<pair<int, int>*> chk_c(c, 1);
    checked_array_iterator<pair<int, int>*>::pointer p_c = &(*chk_c);

    cout << "b[0] = " << *p << endl;
    cout << "c[0].first = " << p_c->first << endl;
}
/* Output:
0
1
2
3
4
b[0] = 0
c[0].first = 10
*/
```

checked_array_iterator::operator->

Returns a pointer to the element addressed by the `checked_array_iterator`.

```
pointer operator->() const;
```

Return Value

A pointer to the element addressed by the `checked_array_iterator` .

Remarks

See [checked_array_iterator::pointer](#) for a code sample.

For more information, see [Checked Iterators](#).

checked_array_iterator::operator++

Increments the `checked_array_iterator` to the next element.

```
checked_array_iterator& operator++();  
  
checked_array_iterator<_Iterator> operator++(int);
```

Return Value

The first operator returns the preincremented `checked_array_iterator` and the second, the postincrement operator, returns a copy of the incremented `checked_array_iterator` .

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_op_plus_plus.cpp  
// compile with: /EHsc  
#include <vector>  
#include <iostream>  
  
int main() {  
    using namespace stdext;  
    using namespace std;  
    int a[] = {6, 3, 77, 199, 222};  
    int b[5];  
    copy(a, a + 5, checked_array_iterator<int*>(b,5));  
  
    checked_array_iterator<int*> checked_output_iterator(b,5);  
  
    cout << *checked_output_iterator << endl;  
    ++checked_output_iterator;  
    cout << *checked_output_iterator << endl;  
    checked_output_iterator++;  
    cout << *checked_output_iterator << endl;  
}  
/* Output:  
6  
3  
77  
*/
```

checked_array_iterator::operator--

Decrements the `checked_array_iterator` to the previous element.

```
checked_array_iterator<_Iterator>& operator--();  
  
checked_array_iterator<_Iterator> operator--(int);
```

Return Value

The first operator returns the predecremented `checked_array_iterator` and the second, the postdecrement operator, returns a copy of the decremented `checked_array_iterator`.

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_op_minus_minus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace stdext;
    using namespace std;
    int a[] = {6, 3, 77, 199, 222};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);

    cout << *checked_output_iterator << endl;
    checked_output_iterator++;
    cout << *checked_output_iterator << endl;
    checked_output_iterator--;
    cout << *checked_output_iterator << endl;
}
/* Output:
6
3
6
*/
```

checked_array_iterator::operator+=

Adds a specified offset to a `checked_array_iterator`.

```
checked_array_iterator<_Iterator>& operator+=(difference_type _Off);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `checked_array_iterator`.

Remarks

For more information, see [Checked Iterators](#).

Example

```

// checked_array_iterators_op_plus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace stdext;
    using namespace std;
    int a[] = {6, 3, 77, 199, 222};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);

    cout << *checked_output_iterator << endl;
    checked_output_iterator += 3;
    cout << *checked_output_iterator << endl;
}
/* Output:
6
199
*/

```

checked_array_iterator::operator+

Adds an offset to an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.

```
checked_array_iterator<_Iterator> operator+(difference_type _Off) const;
```

Parameters

_Off

The offset to be added to the `checked_array_iterator`.

Return Value

A `checked_array_iterator` addressing the offset element.

Remarks

For more information, see [Checked Iterators](#).

Example

```

// checked_array_iterators_op_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace stdext;
    using namespace std;
    int a[] = {6, 3, 77, 199, 222};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);

    cout << *checked_output_iterator << endl;
    checked_output_iterator = checked_output_iterator + 3;
    cout << *checked_output_iterator << endl;
}
/* Output:
6
199
*/

```

checked_array_iterator::operator-=

Decrements a specified offset from a `checked_array_iterator`.

```
checked_array_iterator<_Iterator>& operator-=(difference_type _Off);
```

Parameters

_Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `checked_array_iterator`.

Remarks

For more information, see [Checked Iterators](#).

Example


```
// checked_array_iterators_op_minus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace stdext;
    using namespace std;
    int a[] = {6, 3, 77, 199, 222};
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);

    checked_output_iterator += 3;
    cout << *checked_output_iterator << endl;
    checked_output_iterator -= 2;
    cout << *checked_output_iterator << endl;
}
/* Output:
199
3
*/
```

checked_array_iterator::operator-

Decrements an offset from an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.

```
checked_array_iterator<_Iterator> operator-(difference_type _Off) const;

difference_type operator-(const checked_array_iterator& right) const;
```

Parameters

_Off

The offset to be decremented from the `checked_array_iterator`.

Return Value

A `checked_array_iterator` addressing the offset element.

Remarks

For more information, see [Checked Iterators](#).

checked_array_iterator::operator[]

Returns a reference to an element offset from the element addressed by a `checked_array_iterator` by a specified number of positions.

```
reference operator[](difference_type _Off) const;
```

Parameters

_Off

The offset from the `checked_array_iterator` address.

Return Value

The reference to the element offset.

Remarks

For more information, see [Checked Iterators](#).

Example

```
// checked_array_iterators_op_diff.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace std;
    int V1[10];

    for (int i = 0; i < 10 ; i++)
        V1[i] = i;

    // Declare a difference type for a parameter
    stdext::checked_array_iterator<int*>::difference_type diff = 2;

    stdext::checked_array_iterator<int*> VChkIter(V1, 10);

    stdext::checked_array_iterator<int*>::reference refrpos = VChkIter [diff];

    cout << refrpos + 1 << endl;
}
/* Output:
3
*/
```

checked_array_iterator::pointer

A type that provides a pointer to an element addressed by a `checked_array_iterator`.

```
typedef typename iterator_traits<_Iterator>::pointer pointer;
```

Remarks

See [checked_array_iterator::operator*](#) for a code sample.

For more information, see [Checked Iterators](#).

checked_array_iterator::reference

A type that provides a reference to an element addressed by a `checked_array_iterator`.

```
typedef typename iterator_traits<_Iterator>::reference reference;
```

Remarks

See [checked_array_iterator::operator\[\]](#) for a code sample.

For more information, see [Checked Iterators](#).

See also

[<iterator>](#)

[C++ Standard Library Reference](#)

forward_iterator_tag Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that provides a return type for **iterator_category** function that represents a forward iterator.

Syntax

```
struct forward_iterator_tag      : public input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find out what is the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits < Iterator > ::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as `iterator< Iter> ::iterator_category` when `Iter` describes an object that can serve as a forward iterator.

Example

See [iterator_traits](#) or [random_access_iterator_tag](#) for an example of how to use the **iterator_tags**.

Requirements

Header: `<iterator>`

Namespace: `std`

See also

[input_iterator_tag Struct](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

front_insert_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the front of a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The `front_insert_iterator` class is templated on the type of container.

Syntax

```
template <class Container>
class front_insert_iterator;
```

Parameters

Container

The type of container into the front of which elements are to be inserted by a `front_insert_iterator`.

Remarks

The container must satisfy the requirements for a front insertion sequence where it is possible to insert elements at the beginning of the sequence in amortized constant time. The C++ Standard Library sequence containers defined by the [deque Class](#) and [list Class](#) provide the needed `push_front` member function and satisfy these requirements. By contrast, sequence containers defined by the [vector Class](#) do not satisfy these requirements and cannot be adapted to use with `front_insert_iterator`s. A `front_insert_iterator` must always be initialized with its container.

Constructors

CONSTRUCTOR	DESCRIPTION
front_insert_iterator	Creates an iterator that can insert elements at the front of a specified container object.

Typedefs

TYPE NAME	DESCRIPTION
container_type	A type that represents the container into which a front insertion is to be made.
reference	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

OPERATOR	DESCRIPTION
operator*	Dereferencing operator used to implement the output iterator expression <code>* i = x</code> for a front insertion.

OPERATOR	DESCRIPTION
<code>operator++</code>	Increments the <code>front_insert_iterator</code> to the next location into which a value may be stored.
<code>operator=</code>	Assignment operator used to implement the output iterator expression <code>* i = x</code> for a front insertion.

Requirements

Header: `<iterator>`

Namespace: `std`

`front_insert_iterator::container_type`

A type that represents the container into which a front insertion is to be made.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter *Container*.

Example

```
// front_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L1;
    front_insert_iterator<list<int> >::container_type L2 = L1;
    front_inserter ( L2 ) = 20;
    front_inserter ( L2 ) = 10;
    front_inserter ( L2 ) = 40;

    list<int>::iterator vIter;
    cout << "The list L2 is: ( ";
    for ( vIter = L2.begin ( ) ; vIter != L2.end ( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;
}
/* Output:
The list L2 is: ( 40 10 20 ).
*/
```

`front_insert_iterator::front_insert_iterator`

Creates an iterator that can insert elements at the front of a specified container object.

```
explicit front_insert_iterator(Container& _Cont);
```

Parameters

`_Cont`

The container object into which the `front_insert_iterator` is to insert elements.

Return Value

A `front_insert_iterator` for the parameter container object.

Example

```
// front_insert_iterator_front_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for (i = -1 ; i < 9 ; ++i )
    {
        L.push_back ( 2 * i );
    }

    cout << "The list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;

    // Using the member function to insert an element
    front_inserter ( L ) = 20;

    // Alternatively, one may use the template function
    front_insert_iterator< list < int> > Iter(L);
    *Iter = 30;

    cout << "After the front insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
/* Output:
The list L is:
( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
*/
```

`front_insert_iterator::operator*`

Dereferences the insert iterator returning the element it addresses.

```
front_insert_iterator<Container>& operator*();
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter = value**. If `Iter` is an iterator that addresses an element in a sequence, then ***Iter = value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```
// front_insert_iterator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for ( i = -1 ; i < 9 ; ++i )
    {
        L.push_back ( 2 * i );
    }

    cout << "The list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;

    front_insert_iterator< list< int> > Iter(L);
    *Iter = 20;

    // Alternatively, you may use
    front_inserter ( L ) = 30;

    cout << "After the front insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
/* Output:
The list L is:
( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
*/
```

front_insert_iterator::operator++

Increments the `back_insert_iterator` to the next location into which a value may be stored.

```
front_insert_iterator<Container>& operator++();

front_insert_iterator<Container> operator++(int);
```

Return Value

A `front_insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```

// front_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L1;
    front_insert_iterator<list<int> > iter ( L1 );
    *iter = 10;
    iter++;
    *iter = 20;
    iter++;
    *iter = 30;
    iter++;

    list<int>::iterator vIter;
    cout << "The list L1 is: ( ";
    for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}
/* Output:
The list L1 is: ( 30 20 10 ).
*/

```

front_insert_iterator::operator=

Appends (pushes) a value onto the front of the container.

```

front_insert_iterator<Container>& operator=(typename Container::const_reference val);

front_insert_iterator<Container>& operator=(typename Container::value_type&& val);

```

Parameters

val

The value to be assigned to the container.

Return Value

A reference to the last element inserted at the front of the container.

Remarks

The first member operator evaluates `container.push_front(val)`, then returns `*this`.

The second member operator evaluates

```
container->push_front((typename Container::value_type&& val) ,
```

then returns `*this`.

Example


```

// front_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L1;
    front_insert_iterator<list<int> > iter ( L1 );
    *iter = 10;
    iter++;
    *iter = 20;
    iter++;
    *iter = 30;
    iter++;

    list<int>::iterator vIter;
    cout << "The list L1 is: ( ";
    for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}
/* Output:
The list L1 is: ( 30 20 10 ).
*/

```

front_insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

```

typedef typename Container::reference reference;

```

Example

```

// front_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L;
    front_insert_iterator<list<int> > fiivIter( L );
    *fiivIter = 10;
    *fiivIter = 20;
    *fiivIter = 30;

    list<int>::iterator LIter;
    cout << "The list L is: ( ";
    for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++)
        cout << *LIter << " ";
    cout << ")." << endl;

    front_insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
    cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
/* Output:
The list L is: ( 30 20 10 ).
The first element in the list L is: 30.
*/

```

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

input_iterator_tag Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that provides a return type for `iterator_category` function that represents an input iterator.

Syntax

```
struct input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits < Iterator > ::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as `iterator< Iter> ::iterator_category` when `Iter` describes an object that can serve as an input iterator.

Example

See [iterator_traits](#) or [random_access_iterator_tag](#) for an example of how to use `iterator_tag` S.

Requirements

Header: `<iterator>`

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

insert_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence and associative containers. The `insert_iterator` class is templated on the type of container being adapted.

Syntax

```
template <class Container>
class insert_iterator;
```

Parameters

Container

The type of container into which elements are to be inserted by an `insert_iterator`.

Remarks

The container of type `Container` must satisfy the requirements for a variable-sized container and have a two-argument insert member function where the parameters are of type `Container::iterator` and `Container::value_type` and that returns a type `Container::iterator`. C++ Standard Library sequence and sorted associative containers satisfy these requirements and can be adapted to use with `insert_iterator`s. For associative containers, the position argument is treated as a hint, which has the potential to improve or degrade performance depending on how good the hint is. An `insert_iterator` must always be initialized with its container.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>insert_iterator</code>	Constructs an <code>insert_iterator</code> that inserts an element into a specified position in a container.

Typedefs

TYPE NAME	DESCRIPTION
<code>container_type</code>	A type that represents the container into which a general insertion is to be made.
<code>reference</code>	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	Dereferencing operator used to implement the output iterator expression <code>*i = x</code> for a general insertion.

OPERATOR	DESCRIPTION
<code>operator++</code>	Increments the <code>insert_iterator</code> to the next location into which a value may be stored.
<code>operator=</code>	Assignment operator used to implement the output iterator expression <code>* i = x</code> for a general insertion.

Requirements

Header: `<iterator>`

Namespace: `std`

`insert_iterator::container_type`

A type that represents the container into which a general insertion is to be made.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter *Container*.

Example

```
// insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L1;
    insert_iterator<list<int> >::container_type L2 = L1;
    inserter ( L2, L2.end ( ) ) = 20;
    inserter ( L2, L2.end ( ) ) = 10;
    inserter ( L2, L2.begin ( ) ) = 40;

    list<int>::iterator vIter;
    cout << "The list L2 is: ( ";
    for ( vIter = L2.begin ( ) ; vIter != L2.end ( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}
/* Output:
The list L2 is: ( 40 20 10 ).
*/
```

`insert_iterator::insert_iterator`

Constructs an `insert_iterator` that inserts an element into a specified position in a container.

```
insert_iterator(Container& _Cont, typename Container::iterator _It);
```

Parameters

_Cont

The container into which the `insert_iterator` is to insert elements.

_It

The position for the insertion.

Remarks

All containers have the `insert` member function called by the `insert_iterator`. For associative containers the position parameter is merely a suggestion. The `inserter` function provides a convenient way to insert to values.

Example

```
// insert_iterator_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for (i = 1 ; i < 4 ; ++i )
    {
        L.push_back ( 10 * i );
    }

    cout << "The list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
        cout << *L_Iter << " ";
    cout << ")." << endl;

    // Using the member function to insert an element
    inserter ( L, L.begin( ) ) = 2;

    // Alternatively, you may use the template version
    insert_iterator< list< int> > Iter(L, L.end( ) );
    *Iter = 300;

    cout << "After the insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
/* Output:
The list L is:
( 10 20 30 ).
After the insertions, the list L is:
( 2 10 20 30 300 ).
*/
```

`insert_iterator::operator*`

Dereferences the insert iterator returning the element it addresses.

```
insert_iterator<Container>& operator*();
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter = value**. If `Iter` is an iterator that addresses an element in a sequence, then ***Iter = value** replaces that element with value and does not change the total number of elements in the sequence.

Example

```
// insert_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for (i = 0 ; i < 4 ; ++i )
    {
        L.push_back ( 2 * i );
    }

    cout << "The original list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;

    insert_iterator< list< int> > Iter(L, L.begin ( ) );
    *Iter = 10;
    *Iter = 20;
    *Iter = 30;

    cout << "After the insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
/* Output:
The original list L is:
( 0 2 4 6 ).
After the insertions, the list L is:
( 10 20 30 0 2 4 6 ).
*/
```

insert_iterator::operator++

Increments the `insert_iterator` to the next location into which a value may be stored.

```
insert_iterator<Container>& operator++();

insert_iterator<Container> operator++(int);
```

Parameters

A `insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

```
// insert_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 5 ; ++i )
    {
        vec.push_back ( i );
    }

    vector <int>::iterator vIter;
    cout << "The vector vec is:\n ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    insert_iterator<vector<int> > ii ( vec, vec.begin ( ) );
    *ii = 30;
    ii++;
    *ii = 40;
    ii++;
    *ii = 50;

    cout << "After the insertions, the vector vec becomes:\n ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}
/* Output:
The vector vec is:
( 1 2 3 4 ).
After the insertions, the vector vec becomes:
( 30 40 50 1 2 3 4 ).
*/
```

insert_iterator::operator=

Inserts a value into the container and returns the iterator updated to point to the new element.

```
insert_iterator<Container>& operator=(
    typename Container::const_reference val,);

insert_iterator<Container>& operator=(
    typename Container::value_type&& val);
```

Parameters

val

The value to be assigned to the container.

Return Value

A reference to the element inserted into the container.

Remarks

The first member operator evaluates

```
Iter = container->insert(Iter, val);
```

```
++Iter;
```

then returns `*this`.

The second member operator evaluates

```
Iter = container->insert(Iter, std::move(val));
```

```
++Iter;
```

then returns `*this`.

Example

```
// insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    list<int>::iterator L_Iter;

    list<int> L;
    for (i = 0 ; i < 4 ; ++i )
    {
        L.push_back ( 2 * i );
    }

    cout << "The original list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ) ; L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;

    insert_iterator< list<int> > Iter(L, L.begin( ) );
    *Iter = 10;
    *Iter = 20;
    *Iter = 30;

    cout << "After the insertions, the list L is:\n ( ";
    for ( L_Iter = L.begin( ) ; L_Iter != L.end( ) ; L_Iter++ )
        cout << *L_Iter << " ";
    cout << ")." << endl;
}
/* Output:
The original list L is:
( 0 2 4 6 ).
After the insertions, the list L is:
( 10 20 30 0 2 4 6 ).
*/
```

insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

```
// insert_iterator_container_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    list<int> L;
    insert_iterator<list<int> > iivIter( L , L.begin ( ) );
    *iivIter = 10;
    *iivIter = 20;
    *iivIter = 30;

    list<int>::iterator LIter;
    cout << "The list L is: ( ";
    for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++ )
        cout << *LIter << " ";
    cout << ")." << endl;

    insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
    cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
/* Output:
The list L is: ( 10 20 30 ).
The first element in the list L is: 10.
*/
```

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

istream_iterator Class

10/31/2018 • 5 minutes to read • [Edit Online](#)

Describes an input iterator object. It extracts objects of class `Type` from an input stream, which it accesses through an object it stores, of type `pointer` to `basic_istream` < `CharType`, `Traits` >.

Syntax

```
template <class Type, class CharType = char, class Traits = char_traits<CharType>, class Distance = ptrdiff_t,>
class istream_iterator
: public iterator<
    input_iterator_tag, Type, Distance,
    const Type *,
    const Type&>;
```

Parameters

Type

The type of object to be extracted from the input stream.

CharType

The type that represents the character type for the `istream_iterator`. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the `istream_iterator`. This argument is optional and the default value is `char_traits` < `CharType` >.

Distance

A signed integral type that represents the difference type for the `istream_iterator`. This argument is optional and the default value is `ptrdiff_t`.

After constructing or incrementing an object of class `istream_iterator` with a nonnull stored pointer, the object attempts to extract and store an object of type `Type` from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>istream_iterator</code>	Constructs either an end-of-stream iterator as the default <code>istream_iterator</code> or a <code>istream_iterator</code> initialized to the iterator's stream type from which it reads.

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that provides for the character type of the <code>istream_iterator</code> .

TYPE NAME	DESCRIPTION
<code>istream_type</code>	A type that provides for the stream type of the <code>istream_iterator</code> .
<code>traits_type</code>	A type that provides for the character traits type of the <code>istream_iterator</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	The dereferencing operator returns the stored object of type <code>Type</code> addressed by the <code>istream_iterator</code> .
<code>operator-></code>	Returns the value of a member, if any.
<code>operator++</code>	Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

Requirements

Header: `<iterator>`

Namespace: `std`

`istream_iterator::char_type`

A type that provides for the character type of the `istream_iterator`.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

```

// istream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    typedef istream_iterator<int>::char_type CHT1;
    typedef istream_iterator<int>::traits_type CHTR1;

    // Standard iterator interface for reading
    // elements from the input stream:
    cout << "Enter integers separated by spaces & then\n"
         << " any character ( try example: '2 4 f' ): ";

    // istream_iterator for reading int stream
    istream_iterator<int, CHT1, CHTR1> intRead ( cin );

    // End-of-stream iterator
    istream_iterator<int, CHT1, CHTR1> EOFintRead;

    while ( intRead != EOFintRead )
    {
        cout << "Reading:  " << *intRead << endl;
        ++intRead;
    }
    cout << endl;
}

```

istream_iterator::istream_iterator

Constructs either an end-of-stream iterator as the default `istream_iterator` or a `istream_iterator` initialized to the iterator's stream type from which it reads.

```

istream_iterator();

istream_iterator(istream_type& _Istr);

```

Parameters

_Istr

The input stream to be read use to initialize the `istream_iterator`.

Remarks

The First constructor initializes the input stream pointer with a null pointer and creates an end-of-stream iterator. The second constructor initializes the input stream pointer with `&_Istr`, then attempts to extract and store an object of type `Type`.

The end-of-stream iterator can be use to test whether an `istream_iterator` has reached the end of a stream.

Example

```

// istream_iterator_istream_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
    using namespace std;

    // Used in conjunction with copy algorithm
    // to put elements into a vector read from cin
    vector<int> vec ( 4 );
    vector<int>::iterator Iter;

    cout << "Enter 4 integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";
    istream_iterator<int> intvecRead ( cin );

    // Default constructor will test equal to end of stream
    // for delimiting source range of vecor
    copy ( intvecRead , istream_iterator<int>( ) , vec.begin ( ) );
    cin.clear ( );

    cout << "vec = ";
    for ( Iter = vec.begin( ) ; Iter != vec.end( ) ; Iter++ )
        cout << *Iter << " "; cout << endl;
}

```

istream_iterator::istream_type

A type that provides for the stream type of the `istream_iterator`.

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for `basic_istream<CharType, Traits>`.

Example

See [istream_iterator](#) for an example of how to declare and use `istream_type`.

istream_iterator::operator*

The dereferencing operator returns the stored object of type `Type` addressed by the `istream_iterator`.

```
const Type& operator*() const;
```

Return Value

The stored object of type `Type`.

Example

```

// istream_iterator_operator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
    using namespace std;

    cout << "Enter integers separated by spaces & then\n"
         << " a character ( try example: '2 4 6 8 a' ): ";

    // istream_iterator from stream cin
    istream_iterator<int> intRead ( cin );

    // End-of-stream iterator
    istream_iterator<int> EOFintRead;

    while ( intRead != EOFintRead )
    {
        cout << "Reading:  " << *intRead << endl;
        ++intRead;
    }
    cout << endl;
}

```

istream_iterator::operator->

Returns the value of a member, if any.

```
const Type* operator->() const;
```

Return Value

The value of a member, if any.

Remarks

`i->m` is equivalent to `(*i).m`

The operator returns `&*this` .

Example

```

// istream_iterator_operator_vm.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>
#include <complex>

using namespace std;

int main( )
{
    cout << "Enter complex numbers separated by spaces & then\n"
        << " a character pair ( try example: '(1,2) (3,4) (a,b)' ): ";

    // istream_iterator from stream cin
    istream_iterator< complex<double> > intRead ( cin );

    // End-of-stream iterator
    istream_iterator<complex<double> > EOFintRead;

    while ( intRead != EOFintRead )
    {
        cout << "Reading the real part: " << intRead ->real( ) << endl;
        cout << "Reading the imaginary part: " << intRead ->imag( ) << endl;
        ++intRead;
    }
    cout << endl;
}

```

istream_iterator::operator++

Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

```

istream_iterator<Type, CharType, Traits, Distance>& operator++();

istream_iterator<Type, CharType, Traits, Distance> operator++(int);

```

Return Value

The first member operator returns a reference to the incremented object of type `Type` extracted from the input stream and the second member function returns a copy of the object.

Example


```

// istream_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
    using namespace std;

    cout << "Enter integers separated by spaces & then\n"
         << " a character ( try example: '2 4 6 8 a' ): ";

    // istream_iterator from stream cin
    istream_iterator<int> intRead ( cin );

    // End-of-stream iterator
    istream_iterator<int> EOFintRead;

    while ( intRead != EOFintRead )
    {
        cout << "Reading:  " << *intRead << endl;
        ++intRead;
    }
    cout << endl;
}

```

istream_iterator::traits_type

A type that provides for the character traits type of the `istream_iterator`.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter *Traits*.

Example

```

// istream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
    using namespace std;

    typedef istream_iterator<int>::char_type CHT1;
    typedef istream_iterator<int>::traits_type CHTR1;

    // Standard iterator interface for reading
    // elements from the input stream:
    cout << "Enter integers separated by spaces & then\n"
         << " any character ( try example: '10 20 a' ): ";

    // istream_iterator for reading int stream
    istream_iterator<int, CHT1, CHTR1> intRead ( cin );

    // End-of-stream iterator
    istream_iterator<int, CHT1, CHTR1> EOFintRead;

    while ( intRead != EOFintRead )
    {
        cout << "Reading:  " << *intRead << endl;
        ++intRead;
    }
    cout << endl;
}

```

See also

[input_iterator_tag Struct](#)

[iterator Struct](#)

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

istreambuf_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

The template class `istreambuf_iterator` describes an input iterator object that extracts character elements from an input stream buffer, which it accesses through an object it stores, of type pointer to `basic_streambuf` < **CharType**, **Traits**>.

Syntax

```
template <class CharType class Traits = char_traits <CharType>>
class istreambuf_iterator
: public iterator<input_iterator_tag, CharType, typename Traits ::off_type, CharType*, CharType&>
```

Parameters

CharType

The type that represents the character type for the `istreambuf_iterator`.

Traits

The type that represents the character type for the `istreambuf_iterator`. This argument is optional and the default value is `char_traits` < *CharType*>.

Remarks

The `istreambuf_iterator` class must satisfy the requirements for an input iterator.

After constructing or incrementing an object of class `istreambuf_iterator` with a non-null stored pointer, the object effectively attempts to extract and store an object of type *CharType* from the associated input stream. The extraction may be delayed, however, until the object is actually dereferenced or copied. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Constructors

CONSTRUCTOR	DESCRIPTION
istreambuf_iterator	Constructs an <code>istreambuf_iterator</code> that is initialized to read characters from the input stream.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that provides for the character type of the <code>ostreambuf_iterator</code> .
int_type	A type that provides an integer type for an <code>istreambuf_iterator</code> .
istream_type	A type that provides for the stream type of the <code>istream_iterator</code> .

TYPE NAME	DESCRIPTION
<code>streambuf_type</code>	A type that provides for the stream type of the <code>istreambuf_iterator</code> .
<code>traits_type</code>	A type that provides for the character traits type of the <code>istream_iterator</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>equal</code>	Tests for equality between two input stream buffer iterators.

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	The dereferencing operator returns the next character in the stream.
<code>operator++</code>	Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.
<code>operator-></code>	Returns the value of a member, if any.

Requirements

Header: `<iterator>`

Namespace: `std`

`istreambuf_iterator::char_type`

A type that provides for the character type of the `ostreambuf_iterator`.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

Example

```

// istreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
    using namespace std;

    typedef istreambuf_iterator<char>::char_type CHT1;
    typedef istreambuf_iterator<char>::traits_type CHTR1;

    cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

    // istreambuf_iterator for input stream
    istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
    ostreambuf_iterator<char> charOut ( cout );

    // Used in conjunction with replace_copy algorithm
    // to insert into output stream and replace spaces
    // with dot-separators
    replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}

```

istreambuf_iterator::equal

Tests for equivalence between two input stream buffer iterators.

```
bool equal(const istreambuf_iterator<CharType, Traits>& right) const;
```

Parameters

right

The iterator for which to check for equality.

Return Value

true if both `istreambuf_iterator`s are end-of-stream iterators or if neither is an end-of-stream iterator; otherwise **false**.

Remarks

A range is defined by the `istreambuf_iterator` to the current position and the end-of-stream iterator, but since all non-end-of stream iterators are equivalent under the `equal` member function, it is not possible to define any subranges using `istreambuf_iterator`s. The `==` and `!=` operators have the same semantics.

Example

```
// istreambuf_iterator_equal.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
    using namespace std;

    cout << "(Try the example: 'Hello world!'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

    istreambuf_iterator<char> charReadIn1 ( cin );
    istreambuf_iterator<char> charReadIn2 ( cin );

    bool b1 = charReadIn1.equal ( charReadIn2 );

    if (b1)
        cout << "The iterators are equal." << endl;
    else
        cout << "The iterators are not equal." << endl;
}
```

istreambuf_iterator::int_type

A type that provides an integer type for an `istreambuf_iterator`.

```
typedef typename traits_type::int_type int_type;
```

Remarks

The type is a synonym for `Traits::int_type`.

Example

```
// istreambuf_iterator_int_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
    using namespace std;
    istreambuf_iterator<char>::int_type inttype1 = 100;
    cout << "The inttype1 = " << inttype1 << "." << endl;
}
/* Output:
The inttype1 = 100.
*/
```

istreambuf_iterator::istream_type

A type that provides for the stream type of the `istreambuf_iterator`.

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for `basic_istream < CharType, Traits >`.

Example

See [istreambuf_iterator](#) for an example of how to declare and use `istream_type`.

istreambuf_iterator::istreambuf_iterator

Constructs an `istreambuf_iterator` that is initialized to read characters from the input stream.

```
istreambuf_iterator(istreambuf_type* strbuf = 0) throw();
istreambuf_iterator(istream_type& _Istr) throw();
```

Parameters

strbuf

The input stream buffer to which the `istreambuf_iterator` is being attached.

_Istr

The input stream to which the `istreambuf_iterator` is being attached.

Remarks

The first constructor initializes the input stream-buffer pointer with *strbuf*. The second constructor initializes the input stream-buffer pointer with *_Istr*. `rdbuf`, and then eventually attempts to extract and store an object of type `CharType`.

Example

```
// istreambuf_iterator_istreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
    using namespace std;

    // Following declarations will not compile:
    istreambuf_iterator<char>::istream_type &istrm = cin;
    istreambuf_iterator<char>::istreambuf_type *strmbf = cin.rdbuf( );

    cout << "(Try the example: 'Oh what a world!'\n"
         << " then an Enter key to insert into the output,\n"
         << " & use a ctrl-Z Enter key combination to exit): ";
    istreambuf_iterator<char> charReadIn ( cin );
    ostreambuf_iterator<char> charOut ( cout );

    // Used in conjunction with replace_copy algorithm
    // to insert into output stream and replace spaces
    // with hyphen-separators
    replace_copy ( charReadIn , istreambuf_iterator<char>( ),
                  charOut , ' ' , '-' );
}
```

istreambuf_iterator::operator*

The dereferencing operator returns the next character in the stream.

```
CharType operator*() const;
```

Return Value

The next character in the stream.

Example

```
// istreambuf_iterator_operator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
    using namespace std;

    cout << "Type string of characters & enter to output it,\n"
        << " with stream buffer iterators,(try: 'I'll be back.')
```

istreambuf_iterator::operator++

Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.

```
istreambuf_iterator<CharType, Traits>& operator++();
istreambuf_iterator<CharType, Traits> operator++(int);
```

Return Value

An `istreambuf_iterator` or a reference to an `istreambuf_iterator`.

Remarks

The first operator eventually attempts to extract and store an object of type `CharType` from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

Example


```
// istreambuf_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
    using namespace std;

    cout << "Type string of characters & enter to output it,\n"
        << " with stream buffer iterators,(try: 'I'll be back.')

```

istreambuf_iterator::operator->

Returns the value of a member, if any.

```
const Elem* operator->() const;
```

Return Value

The operator returns **&***this**.

istreambuf_iterator::streambuf_type

A type that provides for the stream type of the istreambuf_iterator.

```
typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for `basic_streambuf` < **CharType, Traits**>.

Example

See [istreambuf_iterator](#) for an example of how to declare and use `istreambuf_type`.

istreambuf_iterator::traits_type

A type that provides for the character traits type of the `istream_iterator`.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter *Traits*.

Example

```

// istreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
    using namespace std;

    typedef istreambuf_iterator<char>::char_type CHT1;
    typedef istreambuf_iterator<char>::traits_type CHTR1;

    cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

    // istreambuf_iterator for input stream
    istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
    ostreambuf_iterator<char> charOut ( cout );

    // Used in conjunction with replace_copy algorithm
    // to insert into output stream and replace spaces
    // with dot-separators
    replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}

```

See also

[iterator Struct](#)

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

iterator Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

An empty base struct used to ensure that a user-defined iterator class works properly with `iterator_traits`.

Syntax

```
struct iterator {  
    typedef Category iterator_category;  
    typedef Type value_type;  
    typedef Distance difference_type;  
    typedef Distance distance_type;  
    typedef Pointer pointer;  
    typedef Reference reference;  
};
```

Remarks

The template struct serves as a base type for all iterators. It defines the member types

- `iterator_category` (a synonym for the template parameter `Category`).
- `value_type` (a synonym for the template parameter `Type`).
- `difference_type` (a synonym for the template parameter `Distance`).
- `distance_type` (a synonym for the template parameter `Distance`).
- `pointer` (a synonym for the template parameter `Pointer`).
- `reference` (a synonym for the template parameter `Reference`).

Note that `value_type` should not be a constant type even if `pointer` points at an object of **const** `Type` and `reference` designates an object of **const** `Type`.

Example

See [iterator_traits](#) for an example of how to declare and use the types in the iterator base class.

Requirements

Header: `<iterator>`

Namespace: `std`

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

iterator_traits Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A template helper struct used to specify all the critical type definitions that an iterator should have.

Syntax

```
struct iterator_traits {  
    typedef typename Iterator::iterator_category iterator_category;  
    typedef typename Iterator::value_type value_type;  
    typedef typename Iterator::difference_type difference_type;  
    typedef difference_type distance_type;  
    typedef typename Iterator::pointer pointer;  
    typedef typename Iterator::reference reference;  
};
```

Remarks

The template struct defines the member types

- `iterator_category` : a synonym for `Iterator::iterator_category` .
- `value_type` : a synonym for `Iterator::value_type` .
- `difference_type` : a synonym for `Iterator::difference_type` .
- `distance_type` : a synonym for `Iterator::difference_type` .
- `pointer` : a synonym for `Iterator::pointer` .
- `reference` : a synonym for `Iterator::reference` .

The partial specializations determine the critical types associated with an object pointer of type **Type *** or **const Type ***.

In this implementation you can also use several template functions that do not make use of partial specialization:

```
template <class Category, class Type, class Diff>  
C _Iter_cat(const iterator<Category, Ty, Diff>&);  
  
template <class Ty>  
random_access_iterator_tag _Iter_cat(const Ty *);  
  
template <class Category, class Ty, class Diff>  
Ty *val_type(const iterator<Category, Ty, Diff>&);  
  
template <class Ty>  
Ty *val_type(const Ty *);  
  
template <class Category, class Ty, class Diff>  
Diff *_Dist_type(const iterator<Category, Ty, Diff>&);  
  
template <class Ty>  
ptrdiff_t *_Dist_type(const Ty *);
```

which determine several of the same types more indirectly. You use these functions as arguments on a function

call. Their sole purpose is to supply a useful template class parameter to the called function.

Example

```
// iterator_traits.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <vector>
#include <list>

using namespace std;

template< class it >
void
function( it i1, it i2 )
{
    iterator_traits<it>::iterator_category cat;
    cout << typeid( cat ).name( ) << endl;
    while ( i1 != i2 )
    {
        iterator_traits<it>::value_type x;
        x = *i1;
        cout << x << " ";
        i1++;
    };
    cout << endl;
};

int main( )
{
    vector<char> vc( 10, 'a' );
    list<int> li( 10 );
    function( vc.begin( ), vc.end( ) );
    function( li.begin( ), li.end( ) );
}
/* Output:
struct std::random_access_iterator_tag
a a a a a a a a a a
struct std::bidirectional_iterator_tag
0 0 0 0 0 0 0 0 0 0
*/
```

Requirements

Header: <iterator>

Namespace: std

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

move_iterator Class

3/28/2019 • 5 minutes to read • [Edit Online](#)

Class template `move_iterator` is a wrapper for an iterator. The `move_iterator` provides the same behavior as the iterator it wraps (stores), except it turns the stored iterator's dereference operator into an rvalue reference, turning a copy into a move. For more information about rvalues, see [Rvalue Reference Declarator: &&](#).

Syntax

```
class move_iterator;
```

Remarks

The template class describes an object that behaves like an iterator except when dereferenced. It stores a random-access iterator of type `Iterator`, accessed by way of the member function `base()`. All operations on a `move_iterator` are performed directly on the stored iterator, except that the result of `operator*` is implicitly cast to `value_type&&` to make an rvalue reference.

A `move_iterator` might be capable of operations that are not defined by the wrapped iterator. These operations should not be used.

Constructors

CONSTRUCTOR	DESCRIPTION
move_iterator	The constructor for objects of type <code>move_iterator</code> .

Typedefs

TYPE NAME	DESCRIPTION
iterator_type	A synonym for the template parameter <code>RandomIterator</code> .
iterator_category	A synonym for a longer typename expression of the same name, <code>iterator_category</code> identifies the general abilities of the iterator.
value_type	A synonym for a longer typename expression of the same name, <code>value_type</code> describes what type the iterator elements are.
difference_type	A synonym for a longer typename expression of the same name, <code>difference_type</code> describes the integral type required to express difference values between elements.
pointer	A synonym for template parameter <code>RandomIterator</code> .
reference	A synonym for the <code>rvalue</code> reference <code>value_type&&</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>base</code>	The member function returns the stored iterator wrapped by this <code>move_iterator</code> .

Operators

OPERATOR	DESCRIPTION
<code>move_iterator::operator*</code>	Returns <code>(reference)*base()</code> .
<code>move_iterator::operator++</code>	Increments the stored iterator. Exact behavior depends on whether it is a preincrement or a postincrement operation.
<code>move_iterator::operator--</code>	Decrements the stored iterator. Exact behavior depends on whether it is a predecrement or a postdecrement operation.
<code>move_iterator::operator-></code>	Returns <code>&***this</code> .
<code>move_iterator::operator-</code>	Returns <code>move_iterator(*this) -=</code> by first subtracting the right-hand value from the current position.
<code>move_iterator::operator[]</code>	Returns <code>(reference)*(*this + off)</code> . Allows you to specify an offset from the current base to obtain the value at that location.
<code>move_iterator::operator+</code>	Returns <code>move_iterator(*this) +=</code> the value. Allows you to add an offset to the base to obtain the value at that location.
<code>move_iterator::operator+=</code>	Adds the right-hand value to the stored iterator, and returns <code>*this</code> .
<code>move_iterator::operator-=</code>	Subtracts the right-hand value from the stored iterator, and returns <code>*this</code> .

Requirements

Header: `<iterator>`

Namespace: `std`

`move_iterator::base`

Returns the stored iterator for this `move_iterator`.

```
RandomIterator base() const;
```

Remarks

The member function returns the stored iterator.

`move_iterator::difference_type`

The type `difference_type` is a `move_iterator` `typedef` based on the iterator trait `difference_type`, and can be

used interchangeably with it.

```
typedef typename iterator_traits<RandomIterator>::difference_type difference_type;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::pointer`.

move_iterator::iterator_category

The type `iterator_category` is a `move_iterator` `typedef` based on the iterator trait `iterator_category`, and can be used interchangeably with it.

```
typedef typename iterator_traits<RandomIterator>::iterator_category iterator_category;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::iterator_category`.

move_iterator::iterator_type

The type `iterator_type` is based on the template parameter `RandomIterator` for the class template `move_iterator`, and can be used interchangeably in its place.

```
typedef RandomIterator iterator_type;
```

Remarks

The type is a synonym for the template parameter `RandomIterator`.

move_iterator::move_iterator

Constructs a move iterator. Uses the parameter as the stored iterator.

```
move_iterator();  
explicit move_iterator(RandomIterator right);  
template <class Type>  
move_iterator(const move_iterator<Type>& right);
```

Parameters

right

The iterator to use as the stored iterator.

Remarks

The first constructor initializes the stored iterator with its default constructor. The remaining constructors initialize the stored iterator with `base.base()`.

move_iterator::operator+=

Adds an offset to the stored iterator, so that the stored iterator points to the element at the new current location. The operator then moves the new current element.

```
move_iterator& operator+=(difference_type _Off);
```


Parameters

_Off

An offset to add to the current position to determine the new current position.

Return Value

Returns the new current element.

Remarks

The operator adds *_Off* to the stored iterator. Then returns `*this`.

move_iterator::operator-=

Moves across a specified number of previous elements. This operator subtracts an offset from the stored iterator.

```
move_iterator& operator-=(difference_type _Off);
```

Parameters

Remarks

The operator evaluates `*this += -_Off`. Then returns `*this`.

move_iterator::operator++

Increments the stored iterator that belongs to this `move_iterator`. The current element is accessed by the postincrement operator. The next element is accessed by the preincrement operator.

```
move_iterator& operator++();  
move_iterator operator++(int);
```

Parameters

Remarks

The first (preincrement) operator increments the stored iterator. Then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, evaluates `++*this`. Then returns the copy.

move_iterator::operator+

Returns the iterator position advanced by any number of elements.

```
move_iterator operator+(difference_type _Off) const;
```

Parameters

Remarks

The operator returns `move_iterator(*this) += _Off`.

move_iterator::operator[]

Allows array index access to elements across the range of the `move iterator`.

```
reference operator[](difference_type _Off) const;
```

Parameters

Remarks

The operator returns `(reference)*(*this + _Off)`.

move_iterator::operator--

Pre- and postdecrement member operators perform a decrement on the stored iterator.

```
move_iterator& operator--();  
move_iterator operator--();
```

Parameters

Remarks

The first member operator (predecrement) decrements the stored iterator. Then returns `*this`.

The second (postdecrement) operator makes a copy of `*this`, evaluates `--*this`. Then returns the copy.

move_iterator::operator-

Decrements the stored iterator and returns the indicated value.

```
move_iterator operator-(difference_type _Off) const;
```

Parameters

Remarks

The operator returns `move_iterator(*this) -= _Off`.

move_iterator::operator*

Dereferences the stored iterator and returns the value. This behaves like an `rvalue reference` and performs a move assignment. The operator transfers the current element out of the base iterator. The element that follows becomes the new current element.

```
reference operator*() const;
```

Remarks

The operator returns `(reference)*base()`.

move_iterator::operator->

Like a normal `RandomIterator` `operator->`, it provides access to the fields that belong to the current element.

```
pointer operator->() const;
```

Remarks

The operator returns `&***this`.

move_iterator::pointer

The type `pointer` is a **typedef** based on the random iterator `RandomIterator` for `move_iterator`, and can be used interchangeably.

```
typedef RandomIterator pointer;
```

Remarks

The type is a synonym for `RandomIterator`.

move_iterator::reference

The type `reference` is a **typedef** based on `value_type&&` for `move_iterator`, and can be used interchangeably with `value_type&&`.

```
typedef value_type&& reference;
```

Remarks

The type is a synonym for `value_type&&`, which is an rvalue reference.

move_iterator::value_type

The type `value_type` is a `move_iterator` **typedef** based on the iterator trait `value_type`, and can be used interchangeably with it.

```
typedef typename iterator_traits<RandomIterator>::value_type value_type;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::value_type`.

See also

[<iterator>](#)

[Lvalues and Rvalues](#)

[Move Constructors and Move Assignment Operators \(C++\)](#)

[C++ Standard Library Reference](#)

ostream_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

The template class `ostream_iterator` describes an output iterator object that writes successive elements onto the output stream with the extraction `operator <<`.

Syntax

```
template <class Type class CharType = char class Traits = char_traits <CharType>>
class ostream_iterator
```

Parameters

Type

The type of object to be inserted into the output stream.

CharType

The type that represents the character type for the `ostream_iterator`. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the `ostream_iterator`. This argument is optional and the default value is `char_traits < CharType >`.

The `ostream_iterator` class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an `ostream_iterator`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>ostream_iterator</code>	Constructs an <code>ostream_iterator</code> that is initialized and delimited to write to the output stream.

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that provides for the character type of the <code>ostream_iterator</code> .
<code>ostream_type</code>	A type that provides for the stream type of the <code>ostream_iterator</code> .
<code>traits_type</code>	A type that provides for the character traits type of the <code>ostream_iterator</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	Dereferencing operator used to implement the output iterator expression <code>* i = x</code> .
<code>operator++</code>	A nonfunctional increment operator that returns an <code>ostream_iterator</code> to the same object it addressed before the operation was called.
<code>operator=</code>	Assignment operator used to implement the output iterator expression <code>* i = x</code> for writing to an output stream.

Requirements

Header: `<iterator>`

Namespace: `std`

`ostream_iterator::char_type`

A type that provides for the character type of the iterator.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

```

// ostream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    typedef ostream_iterator<int>::char_type CHT1;
    typedef ostream_iterator<int>::traits_type CHTR1;

    // ostream_iterator for stream cout
    // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

    // Standard iterator interface for writing
    // elements to the output stream:
    cout << "The integers written to the output stream\n"
        << "by intOut are:" << endl;
    *intOut = 10;
    *intOut = 20;
    *intOut = 30;
}
/* Output:
The integers written to the output stream
by intOut are:
10
20
30
*/

```

ostream_iterator::operator*

Dereferencing operator used to implement the output iterator expression $*\tilde{i} = x$.

```
ostream_iterator<Type, CharType, Traits>& operator*();
```

Return Value

A reference to the `ostream_iterator`.

Remarks

The requirements for an output iterator that the `ostream_iterator` must satisfy require only the expression $*\tilde{i} = t$ be valid and says nothing about the **operator** or the `operator=` on their own. The member operator in this implementation returns ***this**.

Example

```

// ostream_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostream_iterator for stream cout
    // with new line delimiter
    ostream_iterator<int> intOut ( cout , "\n" );

    // Standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *intOut = 10;
    intOut++;           // No effect on iterator position
    *intOut = 20;
    *intOut = 30;
}
/* Output:
Elements written to output stream:
10
20
30
*/

```

ostream_iterator::operator++

A nonfunctional increment operator that returns an `ostream_iterator` to the same object it addressed before the operation was called.

```

ostream_iterator<Type, CharType, Traits>& operator++();
ostream_iterator<Type, CharType, Traits> operator++(int);

```

Return Value

A reference to the `ostream_iterator`.

Remarks

These member operators both return ***this**.

Example

```

// ostream_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostream_iterator for stream cout
    // with new line delimiter
    ostream_iterator<int> intOut ( cout , "\n" );

    // standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *intOut = 10;
    intOut++; // No effect on iterator position
    *intOut = 20;
    *intOut = 30;
}
/* Output:
Elements written to output stream:
10
20
30
*/

```

ostream_iterator::operator=

Assignment operator used to implement the output_iterator expression * `i` = `x` for writing to an output stream.

```
ostream_iterator<Type, CharType, Traits>& operator=(const Type& val);
```

Parameters

val

The value of the object of type `Type` to be inserted into the output stream.

Return Value

The operator inserts *val* into the output stream associated with the object, followed by the delimiter specified in the [ostream_iterator constructor](#) (if any), and then returns a reference to the `ostream_iterator`.

Remarks

The requirements for an output iterator that the `ostream_iterator` must satisfy require only the expression * `ii` = `t` be valid and says nothing about the operator or the operator= on their own. This member operator returns `*this`.

Example


```

// ostream_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostream_iterator for stream cout
    // with new line delimiter
    ostream_iterator<int> intOut ( cout , "\n" );

    // Standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *intOut = 10;
    intOut++;      // No effect on iterator position
    *intOut = 20;
    *intOut = 30;
}
/* Output:
Elements written to output stream:
10
20
30
*/

```

ostream_iterator::ostream_iterator

Constructs an `ostream_iterator` that is initialized and delimited to write to the output stream.

```

ostream_iterator(
    ostream_type& _Ostr);

ostream_iterator(
    ostream_type& _Ostr,
    const CharType* _Delimiter);

```

Parameters

_Ostr

The output stream of type `ostream_iterator::ostream_type` to be iterated over.

_Delimiter

The delimiter that is inserted into the output stream between values.

Remarks

The first constructor initializes the output stream pointer with `&_ostr`. The delimiter string pointer designates an empty string.

The second constructor initializes the output stream pointer with `&_ostr` and the delimiter string pointer with *_Delimiter*.

Example

```

// ostream_iterator_ostream_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostream_iterator for stream cout
    ostream_iterator<int> intOut ( cout , "\n" );
    *intOut = 10;
    intOut++;
    *intOut = 20;
    intOut++;

    int i;
    vector<int> vec;
    for ( i = 1 ; i < 7 ; ++i )
    {
        vec.push_back ( i );
    }

    // Write elements to standard output stream
    cout << "Elements output without delimiter: ";
    copy ( vec.begin ( ), vec.end ( ),
           ostream_iterator<int> ( cout ) );
    cout << endl;

    // Write elements with delimiter " : " to output stream
    cout << "Elements output with delimiter: ";
    copy ( vec.begin ( ), vec.end ( ),
           ostream_iterator<int> ( cout, " : " ) );
    cout << endl;
}
/* Output:
10
20
Elements output without delimiter: 123456
Elements output with delimiter: 1 : 2 : 3 : 4 : 5 : 6 :
*/

```

ostream_iterator::ostream_type

A type that provides for the stream type of the iterator.

```
typedef basic_ostream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for `basic_ostream< CharType , Traits >`, a stream class of the iostream hierarchy that defines objects that can be used for writing.

Example

See [ostream_iterator](#) for an example of how to declare and use `ostream_type`.

ostream_iterator::traits_type

A type that provides for the character traits type of the iterator.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter `Traits`.

Example

```
// ostream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // The following not OK, but are just the default values:
    typedef ostream_iterator<int>::char_type CHT1;
    typedef ostream_iterator<int>::traits_type CHTR1;

    // ostream_iterator for stream cout
    // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

    // Standard iterator interface for writing
    // elements to the output stream:
    cout << "The integers written to output stream\n"
        << "by intOut are:" << endl;
    *intOut = 1;
    *intOut = 10;
    *intOut = 100;
}
/* Output:
The integers written to output stream
by intOut are:
1
10
100
*/
```

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

ostreambuf_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

The template class `ostreambuf_iterator` describes an output iterator object that writes successive character elements onto the output stream with the extraction **operator**`>>`. The `ostreambuf_iterator`s differ from those of the [ostream_iterator Class](#) in having characters instead of a generic type at the type of object being inserted into the output stream.

Syntax

```
template <class CharType = char class Traits = char_traits <CharType>>
```

Parameters

CharType

The type that represents the character type for the `ostreambuf_iterator`. This argument is optional and the default value is **char**.

Traits

The type that represents the character type for the `ostreambuf_iterator`. This argument is optional and the default value is `char_traits < CharType>`.

Remarks

The `ostreambuf_iterator` class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an `ostreambuf_iterator`. The class provides a low-level stream iterator that allows access to the raw (unformatted) I/O stream in the form of characters and the ability to bypass the buffering and character translations associated with the high-level stream iterators.

Constructors

CONSTRUCTOR	DESCRIPTION
ostreambuf_iterator	Constructs an <code>ostreambuf_iterator</code> that is initialized to write characters to the output stream.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that provides for the character type of the <code>ostreambuf_iterator</code> .
ostream_type	A type that provides for the stream type of the <code>ostream_iterator</code> .
streambuf_type	A type that provides for the stream type of the <code>ostreambuf_iterator</code> .

TYPE NAME	DESCRIPTION
<code>traits_type</code>	A type that provides for the character traits type of the <code>ostream_iterator</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>failed</code>	Tests for failure of an insertion into the output stream buffer.

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	Dereferencing operator used to implement the output iterator expression <code>*i = x</code> .
<code>operator++</code>	A nonfunctional increment operator that returns an <code>ostreambuf_iterator</code> to the same object it addressed before the operation was called.
<code>operator=</code>	The operator inserts a character into the associated stream buffer.

Requirements

Header: `<iterator>`

Namespace: `std`

ostreambuf_iterator::char_type

A type that provides for the character type of the `ostreambuf_iterator`.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

```

// ostreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    typedef ostreambuf_iterator<char>::char_type CHT1;
    typedef ostreambuf_iterator<char>::traits_type CHTR1;

    // ostreambuf_iterator for stream cout
    // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

    // Standard iterator interface for writing
    // elements to the output streambuf:
    cout << "The characters written to the output stream\n"
         << " by charOutBuf are: ";
    *charOutBuf = 'O';
    charOutBuf++;
    *charOutBuf = 'U';
    charOutBuf++;
    *charOutBuf = 'T';
    charOutBuf++;
    cout << "." << endl;
}
/* Output:
The characters written to the output stream
by charOutBuf are: OUT.
*/

```

ostreambuf_iterator::failed

Tests for failure of an insertion into the output stream buffer.

```
bool failed() const throw();
```

Return Value

true if no insertion into the output stream buffer has failed earlier; otherwise **false**.

Remarks

The member function returns **true** if, in any prior use of member `operator=`, the call to **subf_->** `sputc` returned **eof**.

Example

```

// ostreambuf_iterator_failed.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostreambuf_iterator for stream cout
    ostreambuf_iterator<char> charOut ( cout );

    *charOut = 'a';
    charOut ++;
    *charOut = 'b';
    charOut ++;
    *charOut = 'c';
    cout << " are characters output individually." << endl;

    bool b1 = charOut.failed ( );
    if (b1)
        cout << "At least one insertion failed." << endl;
    else
        cout << "No insertions failed." << endl;
}
/* Output:
abc are characters output individually.
No insertions failed.
*/

```

ostreambuf_iterator::operator*

A nonfunctional dereferencing operator used to implement the output iterator expression $*i = x$.

```
ostreambuf_iterator<CharType, Traits>& operator*();
```

Return Value

The ostreambuf iterator object.

Remarks

This operator functions only in the output iterator expression $*i = x$ to output characters to stream buffer. Applied to an ostreambuf iterator, it returns the iterator; ***iter** returns **iter**,

Example

```

// ostreambuf_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostreambuf_iterator for stream cout
    // with new line delimiter
    ostreambuf_iterator<char> charOutBuf ( cout );

    // Standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *charOutBuf = 'O';
    charOutBuf++; // no effect on iterator position
    *charOutBuf = 'U';
    *charOutBuf = 'T';
}
/* Output:
Elements written to output stream:
OUT
*/

```

ostreambuf_iterator::operator++

A nonfunctional increment operator that returns an ostream iterator to the same character it addressed before the operation was called.

```

ostreambuf_iterator<CharType, Traits>& operator++();
ostreambuf_iterator<CharType, Traits>& operator++(int);

```

Return Value

A reference to the character originally addressed or to an implementation-defined object that is convertible to

`ostreambuf_iterator` < **CharType**, **Traits**>.

Remarks

The operator is used to implement the output iterator expression $*i = x$.

Example


```

// ostreambuf_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostreambuf_iterator for stream cout
    // with new line delimiter
    ostreambuf_iterator<char> charOutBuf ( cout );

    // Standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *charOutBuf = 'O';
    charOutBuf++; // No effect on iterator position
    *charOutBuf = 'U';
    *charOutBuf = 'T';
}
/* Output:
Elements written to output stream:
OUT
*/

```

ostreambuf_iterator::operator=

The operator inserts a character into the associated stream buffer.

```
ostreambuf_iterator<CharType, Traits>& operator=(CharType _Char);
```

Parameters

_Char

The character to be inserted into the stream buffer.

Return Value

A reference to the character inserted into the stream buffer.

Remarks

Assignment operator used to implement the output iterator expression $*i = x$ for writing to an output stream.

Example

```

// ostreambuf_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostreambuf_iterator for stream cout
    // with new line delimiter
    ostreambuf_iterator<char> charOutBuf ( cout );

    // Standard iterator interface for writing
    // elements to the output stream
    cout << "Elements written to output stream:" << endl;
    *charOutBuf = 'O';
    charOutBuf++; // No effect on iterator position
    *charOutBuf = 'U';
    *charOutBuf = 'T';
}
/* Output:
Elements written to output stream:
OUT
*/

```

ostreambuf_iterator::ostreambuf_iterator

Constructs an `ostreambuf_iterator` that is initialized to write characters to the output stream.

```

ostreambuf_iterator(streambuf_type* strbuf) throw();
ostreambuf_iterator(ostream_type& ostr) throw();

```

Parameters

strbuf

The output streambuf object used to initialize the output stream-buffer pointer.

Ostr

The output stream object used to initialize the output stream-buffer pointer.

Remarks

The first constructor initializes the output stream-buffer pointer with *strbuf*.

The second constructor initializes the output stream-buffer pointer with `ostr`. `rdbuf`. The stored pointer must not be a null pointer.

Example

```

// ostreambuf_iteratorOstreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // ostreambuf_iterator for stream cout
    ostreambuf_iterator<char> charOut ( cout );

    *charOut = 'O';
    charOut ++;
    *charOut = 'U';
    charOut ++;
    *charOut = 'T';
    cout << " are characters output individually." << endl;

    ostreambuf_iterator<char> strOut ( cout );
    string str = "These characters are being written to the output stream.\n ";
    copy ( str.begin ( ), str.end ( ), strOut );
}
/* Output:
OUT are characters output individually.
These characters are being written to the output stream.
*/

```

ostreambuf_iterator::ostream_type

A type that provides for the stream type of the `ostream_iterator`.

```
typedef basicOstream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for `basicOstream<CharType, Traits>`.

Example

See [ostreambuf_iterator](#) for an example of how to declare and use `ostream_type`.

ostreambuf_iterator::streambuf_type

A type that provides for the stream type of the `ostreambuf_iterator`.

```
typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for `basic_streambuf<CharType, Traits>`, a stream class for I/O buffers that becomes `streambuf` when specialized to character type `char`.

Example

See [ostreambuf_iterator](#) for an example of how to declare and use `streambuf_type`.

ostreambuf_iterator::traits_type

A type that provides for the character traits type of the `ostream_iterator`.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter `Traits`.

Example

```
// ostreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    typedef ostreambuf_iterator<char>::char_type CHT1;
    typedef ostreambuf_iterator<char>::traits_type CHTR1;

    // ostreambuf_iterator for stream cout
    // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

    // Standard iterator interface for writing
    // elements to the output streambuf:
    cout << "The characters written to the output stream\n"
         << " by charOutBuf are: ";
    *charOutBuf = 'O';
    charOutBuf++;
    *charOutBuf = 'U';
    charOutBuf++;
    *charOutBuf = 'T';
    charOutBuf++;
    cout << "." << endl;
}
/* Output:
The characters written to the output stream
by charOutBuf are: OUT.
*/
```

See also

[`<iterator>`](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

output_iterator_tag Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that provides a return type for `iterator_category` function that represents an output iterator.

Syntax

```
struct output_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits < Iterator > ::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as `iterator < Iter > ::iterator_category` when `Iter` describes an object that can serve as a output iterator.

This tag is not parameterized on the `value_type` or `difference_type` for the iterator, as with the other iterator tags, because output iterators do not have either a `value_type` or a `difference_type`.

Example

See [iterator_traits](#) or [random_access_iterator_tag](#) for an example of how to use `iterator_tag`s.

Requirements

Header: `<iterator>`

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

random_access_iterator_tag Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that provides a return type for `iterator_category` function that represents a random-access iterator.

Syntax

```
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits < Iterator > ::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as `iterator < Iter > ::iterator_category` when `Iter` describes an object that can serve as a random-access iterator.

Example

```

// iterator_rait.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <list>

using namespace std;

int main( )
{
    vector<int> vi;
    vector<char> vc;
    list<char> lc;
    iterator_traits<vector<int>:: iterator>::iterator_category cati;
    iterator_traits<vector<char>:: iterator>::iterator_category catc;
    iterator_traits<list<char>:: iterator>::iterator_category catlc;

    // These are both random-access iterators
    cout << "The type of iterator for vector<int> is "
        << "identified by the tag:\n "
        << typeid ( cati ).name( ) << endl;
    cout << "The type of iterator for vector<char> is "
        << "identified by the tag:\n "
        << typeid ( catc ).name( ) << endl;
    if ( typeid ( cati ) == typeid( catc ) )
        cout << "The iterators are the same." << endl << endl;
    else
        cout << "The iterators are not the same." << endl << endl;

    // But the list iterator is bidirectinal, not random access
    cout << "The type of iterator for list<char> is "
        << "identified by the tag:\n "
        << typeid ( catlc ).name( ) << endl;

    // cout << ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) ) << endl;
    if ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) )
        cout << "The iterators are the same." << endl;
    else
        cout << "The iterators are not the same." << endl;
    // A random-access iterator is a bidirectional iterator.
    cout << ( void* ) dynamic_cast< iterator_traits<list<char>:: iterator>
        ::iterator_category* > ( &catc ) << endl;
}

```

Sample Output

The following output is for x86.

```

The type of iterator for vector<int> is identified by the tag:
    struct std::random_access_iterator_tag
The type of iterator for vector<char> is identified by the tag:
    struct std::random_access_iterator_tag
The iterators are the same.

The type of iterator for list<char> is identified by the tag:
    struct std::bidirectional_iterator_tag
The iterators are not the same.
0012FF3B

```

Requirements

Header: <iterator>

Namespace: `std`

See also

[bidirectional_iterator_tag](#) Struct

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

reverse_iterator Class

3/28/2019 • 18 minutes to read • [Edit Online](#)

The template class is an iterator adaptor that describes a reverse iterator object that behaves like a random-access or bidirectional iterator, only in reverse. It enables the backward traversal of a range.

Syntax

```
template <class RandomIterator>
class reverse_iterator
```

Parameters

RandomIterator The type that represents the iterator to be adapted to operate in reverse.

Remarks

Existing C++ Standard Library containers also define `reverse_iterator` and `const_reverse_iterator` types and have member functions `rbegin` and `rend` that return reverse iterators. These iterators have overwrite semantics. The `reverse_iterator` adaptor supplements this functionality as it offers insert semantics and can also be used with streams.

The `reverse_iterator` that requires a bidirectional iterator must not call any of the member functions `operator++`, `operator+`, `operator--`, `operator-`, or `operator[]`, which may only be used with random-access iterators.

The range of an iterator is $[first, last)$, where the square bracket on the left indicates the inclusion of *first* and the parenthesis on the right indicates the inclusion of elements up to but excluding *last* itself. The same elements are included in the reversed sequence $[\text{rev} - first, \text{rev} - last)$ so that if *last* is the one-past-the-end element in a sequence, then the first element $\text{rev} - first$ in the reversed sequence points to $*(last - 1)$. The identity which relates all reverse iterators to their underlying iterators is:

$$\&*(\text{reverse_iterator}(i)) == \&*(i - 1).$$

In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Constructors

CONSTRUCTOR	DESCRIPTION
<code>reverse_iterator</code>	Constructs a default <code>reverse_iterator</code> or a <code>reverse_iterator</code> from an underlying iterator.

Typedefs

TYPE NAME	DESCRIPTION
-----------	-------------

TYPE NAME	DESCRIPTION
difference_type	A type that provides the difference between two <code>reverse_iterator</code> s referring to elements within the same container.
iterator_type	A type that provides the underlying iterator for a <code>reverse_iterator</code> .
pointer	A type that provides a pointer to an element addressed by a <code>reverse_iterator</code> .
reference	A type that provides a reference to an element addressed by a <code>reverse_iterator</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
base	Recovers the underlying iterator from its <code>reverse_iterator</code> .

Operators

OPERATOR	DESCRIPTION
operator_star	Returns the element that a <code>reverse_iterator</code> addresses.
operator+	Adds an offset to an iterator and returns the new <code>reverse_iterator</code> addressing the inserted element at the new offset position.
operator++	Increments the <code>reverse_iterator</code> to the next element.
operator+=	Adds a specified offset from a <code>reverse_iterator</code> .
operator-	Subtracts an offset from a <code>reverse_iterator</code> and returns a <code>reverse_iterator</code> addressing the element at the offset position.
operator--	Decrements the <code>reverse_iterator</code> to the previous element.
operator-=	Subtracts a specified offset from a <code>reverse_iterator</code> .
operator->	Returns a pointer to the element addressed by the <code>reverse_iterator</code> .
operator[]	Returns a reference to an element offset from the element addressed by a <code>reverse_iterator</code> by a specified number of positions.

Requirements

Header: `<iterator>`

Namespace: std

reverse_iterator::base

Recovers the underlying iterator from its `reverse_iterator`.

```
RandomIterator base() const;
```

Return Value

The iterator underlying the `reverse_iterator`.

Remarks

The identity that relates all reverse iterators to their underlying iterators is:

$$\&*(\text{reverse_iterator}(i)) == \&(i - 1).$$

In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

```

// reverse_iterator_base.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vector<int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ) ; rvIter++ )
        cout << *rvIter << " ";
    cout << ")." << endl;

    vector<int>::iterator pos, bpos;
    pos = find ( vec.begin ( ), vec.end ( ), 6 );
    cout << "The iterator pos points to: " << *pos << "." << endl;

    typedef reverse_iterator<vector<int>::iterator>::iterator_type it_vec_int_type;

    reverse_iterator<it_vec_int_type> rpos ( pos );
    cout << "The reverse_iterator rpos points to: " << *rpos
        << "." << endl;

    bpos = rpos.base ( );
    cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
}

```

reverse_iterator::difference_type

A type that provides the difference between two `reverse_iterator`s referring to elements within the same container.

```
typedef typename iterator_traits<RandomIterator>::difference_type difference_type;
```

Remarks

The `reverse_iterator` difference type is the same as the iterator difference type.

The type is a synonym for the iterator trait typename `iterator_traits < RandomIterator > ::pointer`.

Example

See [reverse_iterator::operator\[\]](#) for an example of how to declare and use `difference_type`.

reverse_iterator::iterator_type

A type that provides the underlying iterator for a `reverse_iterator`.

```
typedef RandomIterator iterator_type;
```

Remarks

The type is a synonym for the template parameter `Iterator`.

Example

See [reverse_iterator::base](#) for an example of how to declare and use `iterator_type`.

reverse_iterator::operator*

Returns the element that a `reverse_iterator` addresses.

```
reference operator*() const;
```

Return Value

The value of the elements addressed by the `reverse_iterator`.

Remarks

The operator returns `*(current - 1)`.

Example

```

// reverse_iterator_op_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++ )
        cout << *rvIter << " ";
    cout << ")." << endl;

    vector <int>::iterator pos, bpos;
    pos = find ( vec.begin ( ), vec.end ( ), 6 );

    // Declare a difference type for a parameter
    // declare a reference return type
    reverse_iterator<vector<int>::iterator>::reference refpos = *pos;
    cout << "The iterator pos points to: " << refpos << "." << endl;
}

```

reverse_iterator::operator+

Adds an offset to an iterator and returns the new `reverse_iterator` addressing the inserted element at the new offset position.

```
reverse_iterator<RandomIterator> operator+(difference_type Off) const;
```

Parameters

Off

The offset to be added to the reverse iterator.

Return Value

A `reverse_iterator` addressing the offset element.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

Example

```

// reverse_iterator_op_add.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the first element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    vector <int>::reverse_iterator rVPOS2 =rVPOS1 + 2; // offset added
    cout << "After the +2 offset, the iterator rVPOS2 points\n"
        << " to the 3rd element in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}

```

The vector vec is: (2 4 6 8 10).
 The vector vec reversed is: (10 8 6 4 2).
 The iterator rVPOS1 initially points to the first element
 in the reversed sequence: 10.
 After the +2 offset, the iterator rVPOS2 points
 to the 3rd element in the reversed sequence: 6.

reverse_iterator::operator++

Increments the reverse_iterator to the previous element.

```

reverse_iterator<RandomIterator>& operator++();
reverse_iterator<RandomIterator> operator++(int);

```

Return Value

The first operator returns the preincremented `reverse_iterator` and the second, the postincrement operator, returns a copy of the incremented `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a bidirectional iterator.

Example

```
// reverse_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i - 1 );
    }

    vector<int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector<int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector<int>::reverse_iterator rVPOS1 = vec.rbegin( );

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    rVPOS1++; // postincrement, preincrement: ++rVPOS1

    cout << "After incrementing, the iterator rVPOS1 points\n"
        << " to the second element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

```
The vector vec is: ( 1 3 5 7 9 ).
The vector vec reversed is: ( 9 7 5 3 1 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 9.
After incrementing, the iterator rVPOS1 points
to the second element in the reversed sequence: 7.
```

reverse_iterator::operator+=

Adds a specified offset from a reverse_iterator.

```
reverse_iterator<RandomIterator>& operator+=(difference_type Off);
```

Parameters

Off

The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `reverse_iterator`.

Example

```
// reverse_iterator_op_addoff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    rVPOS1+=2; // addition of an offset
    cout << "After the +2 offset, the iterator rVPOS1 now points\n"
        << " to the third element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

The vector vec is: (2 4 6 8 10).
The vector vec reversed is: (10 8 6 4 2).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
After the +2 offset, the iterator rVPOS1 now points
to the third element in the reversed sequence: 6.

reverse_iterator::operator-

Subtracts an offset from a `reverse_iterator` and returns a `reverse_iterator` addressing the element at the offset position.

```
reverse_iterator<RandomIterator> operator-(difference_type Off) const;
```

Parameters

Off

The offset to be subtracted from the reverse_iterator.

Return Value

A `reverse_iterator` addressing the offset element.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

Example

```
// reverse_iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 3 * i );
    }

    vector <int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ) ; vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ) ; rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the first element
    vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

    cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    vector <int>::reverse_iterator rVPOS2 =rVPOS1 - 2; // offset subtracted
    cout << "After the -2 offset, the iterator rVPOS2 points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}
```

The vector `vec` is: (3 6 9 12 15).
The vector `vec` reversed is: (15 12 9 6 3).
The iterator `rVPOS1` initially points to the last element
in the reversed sequence: 3.
After the -2 offset, the iterator `rVPOS2` points
to the 2nd element from the last in the reversed sequence: 9.

`reverse_iterator::operator--`

Decrements the `reverse_iterator` to the previous element.

```
reverse_iterator<RandomIterator>& operator--();  
reverse_iterator<RandomIterator> operator--(int);
```

Return Value

The first operator returns the predecremented `reverse_iterator` and the second, the postdecrement operator, returns a copy of the decremented `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a bidirectional iterator.

Example

```

// reverse_iterator_op_decr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i - 1 );
    }

    vector <int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the first element
    vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

    cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
    rVPOS1--; // postdecrement, predecrement: --rVPOS1

    cout << "After the decrement, the iterator rVPOS1 points\n"
        << " to the next-to-last element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}

```

The vector vec is: (1 3 5 7 9).
 The vector vec reversed is: (9 7 5 3 1).
 The iterator rVPOS1 initially points to the last element
 in the reversed sequence: 1.
 After the decrement, the iterator rVPOS1 points
 to the next-to-last element in the reversed sequence: 3.

reverse_iterator::operator-=

Subtracts a specified offset from a `reverse_iterator`.

```
reverse_iterator<RandomIterator>& operator-=(difference_type Off);
```

Parameters

Off

The offset to be subtracted from the `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

The operator evaluates **current** + *_ Off*. then returns ***this**.

Example

```
// reverse_iterator_op_suboff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 3 * i );
    }

    vector <int>::iterator vIter;

    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    // Initializing reverse_iterators to the first element
    vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

    cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    rVPOS1-=2;      // Subtraction of an offset
    cout << "After the -2 offset, the iterator rVPOS1 now points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

```
The vector vec is: ( 3 6 9 12 15 ).
The vector vec reversed is: ( 15 12 9 6 3 ).
The iterator rVPOS1 initially points to the last element
in the reversed sequence: 3.
After the -2 offset, the iterator rVPOS1 now points
to the 2nd element from the last in the reversed sequence: 9.
```

reverse_iterator::operator->

Returns a pointer to the element addressed by the `reverse_iterator` .

```
pointer operator->() const;
```

Return Value

A pointer to the element addressed by the `reverse_iterator`.

Remarks

The operator returns **&*&this**.

Example

```
// reverse_iterator_ptrto.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
    using namespace std;

    typedef vector<pair<int,int> > pVector;
    pVector vec;
    vec.push_back(pVector::value_type(1,2));
    vec.push_back(pVector::value_type(3,4));
    vec.push_back(pVector::value_type(5,6));

    pVector::iterator pvIter;
    cout << "The vector vec of integer pairs is:\n( ";
    for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
        cout << "( " << pvIter -> first << ", " << pvIter -> second << " ) ";
    cout << ")" << endl << endl;

    pVector::reverse_iterator rpvIter;
    cout << "The vector vec reversed is:\n( ";
    for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++ )
        cout << "( " << rpvIter -> first << ", " << rpvIter -> second << " ) ";
    cout << ")" << endl << endl;

    pVector::iterator pos = vec.begin ( );
    pos++;
    cout << "The iterator pos points to:\n( " << pos -> first << ", "
    << pos -> second << " )" << endl << endl;

    pVector::reverse_iterator rpos (pos);

    // Use operator -> with return type: why type int and not int*
    int fint = rpos -> first;
    int sint = rpos -> second;

    cout << "The reverse_iterator rpos points to:\n( " << fint << ", "
    << sint << " )" << endl;
}
```

```
The vector vec of integer pairs is:  
( ( 1, 2) ( 3, 4) ( 5, 6) )
```

```
The vector vec reversed is:  
( ( 5, 6) ( 3, 4) ( 1, 2) )
```

```
The iterator pos points to:  
( 3, 4 )
```

```
The reverse_iterator rpos points to:  
( 1, 2 )
```

reverse_iterator::operator[]

Returns a reference to an element offset from the element addressed by a `reverse_iterator` by a specified number of positions.

```
reference operator[](difference_type Off) const;
```

Parameters

Off

The offset from the `reverse_iterator` address.

Return Value

The reference to the element offset.

Remarks

The operator returns `*(*this + off)`.

Example

```

// reverse_iterator_ret_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for (i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( 2 * i );
    }

    vector <int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ) ; rvIter++ )
        cout << *rvIter << " ";
    cout << ")." << endl;

    vector <int>::iterator pos;
    pos = find ( vec.begin ( ), vec.end ( ), 8 );
    reverse_iterator<vector<int>::iterator> rpos ( pos );

    // Declare a difference type for a parameter
    reverse_iterator<vector<int>::iterator>::difference_type diff = 2;

    cout << "The iterator pos points to: " << *pos << "." << endl;
    cout << "The iterator rpos points to: " << *rpos << "." << endl;

    // Declare a reference return type & use operator[]
    reverse_iterator<vector<int>::iterator>::reference refrpos = rpos [diff];
    cout << "The iterator rpos now points to: " << refrpos << "." << endl;
}

```

```

The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator pos points to: 8.
The iterator rpos points to: 6.
The iterator rpos now points to: 2.

```

reverse_iterator::pointer

A type that provides a pointer to an element addressed by a `reverse_iterator`.

```

typedef typename iterator_traits<RandomIterator>::pointer pointer;

```

Remarks

The type is a synonym for the iterator trait typename `iterator_traits < RandomIterator > ::pointer`.

Example


```

// reverse_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
    using namespace std;

    typedef vector<pair<int,int> > pVector;
    pVector vec;
    vec.push_back( pVector::value_type( 1,2 ) );
    vec.push_back( pVector::value_type( 3,4 ) );
    vec.push_back( pVector::value_type( 5,6 ) );

    pVector::iterator pvIter;
    cout << "The vector vec of integer pairs is:\n" << "( ";
    for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
        cout << "( " << pvIter -> first << ", " << pvIter -> second << " ) ";
    cout << ")" << endl;

    pVector::reverse_iterator rpvIter;
    cout << "\nThe vector vec reversed is:\n" << "( ";
    for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++)
        cout << "( " << rpvIter -> first << ", " << rpvIter -> second << " ) ";
    cout << ")" << endl;

    pVector::iterator pos = vec.begin ( );
    pos++;
    cout << "\nThe iterator pos points to:\n"
        << "( " << pos -> first << ", "
        << pos -> second << " )" << endl;

    pVector::reverse_iterator rpos (pos);
    cout << "\nThe iterator rpos points to:\n"
        << "( " << rpos -> first << ", "
        << rpos -> second << " )" << endl;
}

```

The vector vec of integer pairs is:
 ((1, 2) (3, 4) (5, 6))

The vector vec reversed is:
 ((5, 6) (3, 4) (1, 2))

The iterator pos points to:
 (3, 4)

The iterator rpos points to:
 (1, 2)

reverse_iterator::reference

A type that provides a reference to an element addressed by a reverse_iterator.

```

typedef typename iterator_traits<RandomIterator>::reference reference;

```

Remarks

The type is a synonym for the iterator trait typename `iterator_traits< RandomIterator>::reference`.

Example

See `reverse_iterator::operator[]` or `reverse_iterator::operator*` for examples of how to declare and use `reference`.

reverse_iterator::reverse_iterator

Constructs a default `reverse_iterator` or a `reverse_iterator` from an underlying iterator.

```
reverse_iterator();
explicit reverse_iterator(RandomIterator right);

template <class Type>
reverse_iterator(const reverse_iterator<Type>& right);
```

Parameters

right

The iterator that is to be adapted to a `reverse_iterator`.

Return Value

A default `reverse_iterator` or a `reverse_iterator` adapting an underlying iterator.

Remarks

The identity which relates all reverse iterators to their underlying iterators is:

$$\&*(\text{reverse_iterator}(i)) == \&(i - 1).$$

In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

```

// reverse_iterator_reverse_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    vector<int> vec;
    for ( i = 1 ; i < 6 ; ++i )
    {
        vec.push_back ( i );
    }

    vector <int>::iterator vIter;
    cout << "The vector vec is: ( ";
    for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
        cout << *vIter << " ";
    cout << ")." << endl;

    vector <int>::reverse_iterator rvIter;
    cout << "The vector vec reversed is: ( ";
    for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
        cout << *rvIter << " ";
    cout << ")." << endl;

    vector <int>::iterator pos;
    pos = find ( vec.begin ( ), vec.end ( ), 4 );
    cout << "The iterator pos = " << *pos << "." << endl;

    vector <int>::reverse_iterator rpos ( pos );
    cout << "The reverse_iterator rpos = " << *rpos
        << "." << endl;
}

```

See also

[<iterator>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

unchecked_array_iterator Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `unchecked_array_iterator` class allows you to wrap an array or pointer into an unchecked iterator. Use this class as a wrapper (using the [make_unchecked_array_iterator](#) function) for raw pointers or arrays as a targeted way to manage unchecked pointer warnings instead of globally silencing these warnings. If possible, prefer the checked version of this class, [checked_array_iterator](#).

NOTE

This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

Syntax

```
template <class Iterator>
class unchecked_array_iterator;
```

Remarks

This class is defined in the [stdext](#) namespace.

This is the unchecked version of the [checked_array_iterator Class](#) and supports all the same overloads and members. For more information on the checked iterator feature with code examples, see [Checked Iterators](#).

Requirements

Header: <iterator>

Namespace: stdext

See also

[<iterator>](#)

[C++ Standard Library Reference](#)

<limits>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template class `numeric_limits` and two enumerations concerning floating-point representations and rounding.

Syntax

```
#include <limits>
```

Remarks

Explicit specializations of the `numeric_limits` class describe many properties of the fundamental types, including the character, integer, and floating-point types and **bool** that are implementation defined rather than fixed by the rules of the C++ language. Properties described in <limits> include accuracy, minimum and maximum sized representations, rounding, and signaling type errors.

Enumerations

<code>float_denorm_style</code>	The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:
<code>float_round_style</code>	The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

Classes

CLASS	DESCRIPTION
<code>numeric_limits</code> Class	The template class describes arithmetic properties of built-in numerical types.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<limits> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

[float_denorm_style](#)

[float_round_style](#)

float_denorm_style Enumeration

The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:

```
enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1    };
```

Return Value

The enumeration returns:

- `denorm_indeterminate` if the presence or absence of denormalized forms cannot be determined at translation time.
- `denorm_absent` if denormalized forms are absent.
- `denorm_present` if denormalized forms are present.

Example

See [numeric_limits::has_denorm](#) for an example in which the values of this enumeration may be accessed.

float_round_style Enumeration

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

```
enum float_round_style {
    round_indeterminate = -1,
    round_toward_zero = 0,
    round_to_nearest = 1,
    round_toward_infinity = 2,
    round_toward_neg_infinity = 3    };
```

Return Value

The enumeration returns:

- `round_indeterminate` if the rounding method cannot be determined.
- `round_toward_zero` if the round toward zero.
- `round_to_nearest` if the round to nearest integer.
- `round_toward_infinity` if the round away from zero.

- `round_toward_neg_infinity` if the round to more negative integer.

Example

See [numeric_limits::round_style](#) for an example in which the values of this enumeration may be accessed.

See also

[<limits>](#)

numeric_limits Class

10/31/2018 • 27 minutes to read • [Edit Online](#)

The template class describes arithmetic properties of built-in numerical types.

Syntax

```
template <class Type>
class numeric_limits
```

Parameters

Type

The fundamental element data type whose properties are being tested or queried or set.

Remarks

The header defines explicit specializations for the types **wchar_t**, **bool**, **char**, **signed char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double**, **long double**, **long long**, **unsigned long long**, **char16_t**, and **char32_t**. For these explicit specializations, the member `numeric_limits::is_specialized` is **true**, and all relevant members have meaningful values. The program can supply additional explicit specializations. Most member functions of the class describe or test possible implementations of **float**.

For an arbitrary specialization, no members have meaningful values. A member object that does not have a meaningful value stores zero (or **false**) and a member function that does not return a meaningful value returns

`Type(0)`.

Static Functions and Constants

<code>denorm_min</code>	Returns the smallest nonzero denormalized value.
<code>digits</code>	Returns the number of radix digits that the type can represent without loss of precision.
<code>digits10</code>	Returns the number of decimal digits that the type can represent without loss of precision.
<code>epsilon</code>	Returns the difference between 1 and the smallest value greater than 1 that the data type can represent.
<code>has_denorm</code>	Tests whether a type allows denormalized values.
<code>has_denorm_loss</code>	Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.
<code>has_infinity</code>	Tests whether a type has a representation for positive infinity.
<code>has_quiet_NaN</code>	Tests whether a type has a representation for a quiet not a number (NaN), which is nonsignaling.

<code>has_signaling_NaN</code>	Tests whether a type has a representation for signaling not a number (NaN).
<code>infinity</code>	The representation for positive infinity for a type, if available.
<code>is_bounded</code>	Tests if the set of values that a type may represent is finite.
<code>is_exact</code>	Tests if the calculations done on a type are free of rounding errors.
<code>is_iec559</code>	Tests if a type conforms to IEC 559 standards.
<code>is_integer</code>	Tests if a type has an integer representation.
<code>is_modulo</code>	Tests if a type has a modulo representation.
<code>is_signed</code>	Tests if a type has a signed representation.
<code>is_specialized</code>	Tests if a type has an explicit specialization defined in the template class <code>numeric_limits</code> .
<code>lowest</code>	Returns the most negative finite value.
<code>max</code>	Returns the maximum finite value for a type.
<code>max_digits10</code>	Returns the number of decimal digits required to ensure that two distinct values of the type have distinct decimal representations.
<code>max_exponent</code>	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.
<code>max_exponent10</code>	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.
<code>min</code>	Returns the minimum normalized value for a type.
<code>min_exponent</code>	Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.
<code>min_exponent10</code>	Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.
<code>quiet_NaN</code>	Returns the representation of a quiet not a number (NaN) for the type.
<code>radix</code>	Returns the integral base, referred to as radix, used for the representation of a type.

round_error	Returns the maximum rounding error for the type.
round_style	Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.
signaling_NaN	Returns the representation of a signaling not a number (NaN) for the type.
tinyness_before	Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.
traps	Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

Requirements

Header: <limits>

Namespace: std

numeric_limits::denorm_min

Returns the smallest nonzero denormalized value.

```
static constexpr Type denorm_min() throw();
```

Return Value

The smallest nonzero denormalized value.

Remarks

long double is the same as **double** for the C++ compiler.

The function returns the minimum value for the type, which is the same as [min](#) if [has_denorm](#) is not equal to

`denorm_present` .

Example

```

// numeric_limits_denorm_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The smallest nonzero denormalized value" << endl
         << "for float objects is: "
         << numeric_limits<float>::denorm_min( ) << endl;
    cout << "The smallest nonzero denormalized value" << endl
         << "for double objects is: "
         << numeric_limits<double>::denorm_min( ) << endl;
    cout << "The smallest nonzero denormalized value" << endl
         << "for long double objects is: "
         << numeric_limits<long double>::denorm_min( ) << endl;

    // A smaller value will round to zero
    cout << numeric_limits<float>::denorm_min( )/2 <<endl;
    cout << numeric_limits<double>::denorm_min( )/2 <<endl;
    cout << numeric_limits<long double>::denorm_min( )/2 <<endl;
}

```

```

The smallest nonzero denormalized value
for float objects is: 1.4013e-045
The smallest nonzero denormalized value
for double objects is: 4.94066e-324
The smallest nonzero denormalized value
for long double objects is: 4.94066e-324
0
0
0

```

numeric_limits::digits

Returns the number of radix digits that the type can represent without loss of precision.

```
static constexpr int digits = 0;
```

Return Value

The number of radix digits that the type can represent without loss of precision.

Remarks

The member stores the number of radix digits that the type can represent without change, which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type.

Example

```
// numeric_limits_digits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << numeric_limits<float>::digits <<endl;
    cout << numeric_limits<double>::digits <<endl;
    cout << numeric_limits<long double>::digits <<endl;
    cout << numeric_limits<int>::digits <<endl;
    cout << numeric_limits<__int64>::digits <<endl;
}
```

```
24
53
53
31
63
```

numeric_limits::digits10

Returns the number of decimal digits that the type can represent without loss of precision.

```
static constexpr int digits10 = 0;
```

Return Value

The number of decimal digits that the type can represent without loss of precision.

Example

```
// numeric_limits_digits10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << numeric_limits<float>::digits10 <<endl;
    cout << numeric_limits<double>::digits10 <<endl;
    cout << numeric_limits<long double>::digits10 <<endl;
    cout << numeric_limits<int>::digits10 <<endl;
    cout << numeric_limits<__int64>::digits10 <<endl;
    float f = (float)99999999;
    cout.precision ( 10 );
    cout << "The float is; " << f << endl;
}
```

```
6
15
15
9
18
The float is; 100000000
```

numeric_limits::epsilon

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the data type.

```
static constexpr Type epsilon() throw();
```

Return Value

The difference between 1 and the smallest value greater than 1 that is representable for the data type.

Remarks

The value is FLT_EPSILON for type **float**. `epsilon` for a type is the smallest positive floating-point number N such that $N + \text{epsilon} + N$ is representable.

Example

```
// numeric_limits_epsilon.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for float objects is: "
        << numeric_limits<float>::epsilon( ) << endl;
    cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for double objects is: "
        << numeric_limits<double>::epsilon( ) << endl;
    cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for long double objects is: "
        << numeric_limits<long double>::epsilon( ) << endl;
}
```

```
The difference between 1 and the smallest value greater than 1
for float objects is: 1.19209e-007
The difference between 1 and the smallest value greater than 1
for double objects is: 2.22045e-016
The difference between 1 and the smallest value greater than 1
for long double objects is: 2.22045e-016
```

numeric_limits::has_denorm

Tests whether a type allows denormalized values.

```
static constexpr float_denorm_style has_denorm = denorm_absent;
```

Return Value

An enumeration value of type **const** `float_denorm_style`, indicating whether the type allows denormalized values.

Remarks

The member stores `denorm_present` for a floating-point type that has denormalized values, effectively a variable number of exponent bits.

Example

```
// numeric_limits_has_denorm.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects allow denormalized values: "
          << numeric_limits<float>::has_denorm
          << endl;
    cout << "Whether double objects allow denormalized values: "
          << numeric_limits<double>::has_denorm
          << endl;
    cout << "Whether long int objects allow denormalized values: "
          << numeric_limits<long int>::has_denorm
          << endl;
}
```

```
Whether float objects allow denormalized values: 1
Whether double objects allow denormalized values: 1
Whether long int objects allow denormalized values: 0
```

numeric_limits::has_denorm_loss

Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.

```
static constexpr bool has_denorm_loss = false;
```

Return Value

true if the loss of accuracy is detected as a denormalization loss; **false** if not.

Remarks

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 floating-point representations that can affect some results.

Example

```
// numeric_limits_has_denorm_loss.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects can detect denormalized loss: "
          << numeric_limits<float>::has_denorm_loss
          << endl;
    cout << "Whether double objects can detect denormalized loss: "
          << numeric_limits<double>::has_denorm_loss
          << endl;
    cout << "Whether long int objects can detect denormalized loss: "
          << numeric_limits<long int>::has_denorm_loss
          << endl;
}
```

```
Whether float objects can detect denormalized loss: 1
Whether double objects can detect denormalized loss: 1
Whether long int objects can detect denormalized loss: 0
```

numeric_limits::has_infinity

Tests whether a type has a representation for positive infinity.

```
static constexpr bool has_infinity = false;
```

Return Value

true if the type has a representation for positive infinity; **false** if not.

Remarks

The member returns **true** if [is_iec559](#) is **true**.

Example

```
// numeric_limits_has_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have infinity: "
          << numeric_limits<float>::has_infinity
          << endl;
    cout << "Whether double objects have infinity: "
          << numeric_limits<double>::has_infinity
          << endl;
    cout << "Whether long int objects have infinity: "
          << numeric_limits<long int>::has_infinity
          << endl;
}
```

```
Whether float objects have infinity: 1
Whether double objects have infinity: 1
Whether long int objects have infinity: 0
```

numeric_limits::has_quiet_NaN

Tests whether a type has a representation for a quiet not a number (NaN), which is nonsignaling.

```
static constexpr bool has_quiet_NaN = false;
```

Return Value

true if the **type** has a representation for a quiet NaN; **false** if not.

Remarks

A quiet NaN is an encoding for not a number, which does not signal its presence in an expression. The return value is **true** if [is_iec559](#) is true.

Example

```
// numeric_limits_has_quiet_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have quiet_NaN: "
          << numeric_limits<float>::has_quiet_NaN
          << endl;
    cout << "Whether double objects have quiet_NaN: "
          << numeric_limits<double>::has_quiet_NaN
          << endl;
    cout << "Whether long int objects have quiet_NaN: "
          << numeric_limits<long int>::has_quiet_NaN
          << endl;
}
```

```
Whether float objects have quiet_NaN: 1
Whether double objects have quiet_NaN: 1
Whether long int objects have quiet_NaN: 0
```

numeric_limits::has_signaling_NaN

Tests whether a type has a representation for signaling not a number (NaN).

```
static constexpr bool has_signaling_NaN = false;
```

Return Value

true if the type has a representation for a signaling NaN; **false** if not.

Remarks

A signaling NaN is an encoding for not a number, which signals its presence in an expression. The return value is

true if [is_iec559](#) is true.

Example

```
// numeric_limits_has_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have a signaling_NaN: "
          << numeric_limits<float>::has_signaling_NaN
          << endl;
    cout << "Whether double objects have a signaling_NaN: "
          << numeric_limits<double>::has_signaling_NaN
          << endl;
    cout << "Whether long int objects have a signaling_NaN: "
          << numeric_limits<long int>::has_signaling_NaN
          << endl;
}
```

```
Whether float objects have a signaling_NaN: 1
Whether double objects have a signaling_NaN: 1
Whether long int objects have a signaling_NaN: 0
```

numeric_limits::infinity

The representation of positive infinity for a type, if available.

```
static constexpr Type infinity() throw();
```

Return Value

The representation of positive infinity for a type, if available.

Remarks

The return value is meaningful only if [has_infinity](#) is **true**.

Example

```

// numeric_limits_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << numeric_limits<float>::has_infinity <<endl;
    cout << numeric_limits<double>::has_infinity<<endl;
    cout << numeric_limits<long double>::has_infinity <<endl;
    cout << numeric_limits<int>::has_infinity <<endl;
    cout << numeric_limits<__int64>::has_infinity <<endl;

    cout << "The representation of infinity for type float is: "
         << numeric_limits<float>::infinity( ) <<endl;
    cout << "The representation of infinity for type double is: "
         << numeric_limits<double>::infinity( ) <<endl;
    cout << "The representation of infinity for type long double is: "
         << numeric_limits<long double>::infinity( ) <<endl;
}

```

```

1
1
1
0
0
The representation of infinity for type float is: inf
The representation of infinity for type double is: inf
The representation of infinity for type long double is: inf

```

numeric_limits::is_bounded

Tests if the set of values that a type may represent is finite.

```
static constexpr bool is_bounded = false;
```

Return Value

true if the type has a bounded set of representable values; **false** if not.

Remarks

All predefined types have a bounded set of representable values and return **true**.

Example

```

// numeric_limits_is_bounded.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have bounded set "
          << "of representable values: "
          << numeric_limits<float>::is_bounded
          << endl;
    cout << "Whether double objects have bounded set "
          << "of representable values: "
          << numeric_limits<double>::is_bounded
          << endl;
    cout << "Whether long int objects have bounded set "
          << "of representable values: "
          << numeric_limits<long int>::is_bounded
          << endl;
    cout << "Whether unsigned char objects have bounded set "
          << "of representable values: "
          << numeric_limits<unsigned char>::is_bounded
          << endl;
}

```

```

Whether float objects have bounded set of representable values: 1
Whether double objects have bounded set of representable values: 1
Whether long int objects have bounded set of representable values: 1
Whether unsigned char objects have bounded set of representable values: 1

```

numeric_limits::is_exact

Tests if the calculations done on a type are free of rounding errors.

```
static constexpr bool is_exact = false;
```

Return Value

true if the calculations are free of rounding errors; **false** if not.

Remarks

All predefined integer types have exact representations for their values and return **false**. A fixed-point or rational representation is also considered exact, but a floating-point representation is not.

Example

```

// numeric_limits_is_exact.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have calculations "
          << "free of rounding errors: "
          << numeric_limits<float>::is_exact
          << endl;
    cout << "Whether double objects have calculations "
          << "free of rounding errors: "
          << numeric_limits<double>::is_exact
          << endl;
    cout << "Whether long int objects have calculations "
          << "free of rounding errors: "
          << numeric_limits<long int>::is_exact
          << endl;
    cout << "Whether unsigned char objects have calculations "
          << "free of rounding errors: "
          << numeric_limits<unsigned char>::is_exact
          << endl;
}

```

```

Whether float objects have calculations free of rounding errors: 0
Whether double objects have calculations free of rounding errors: 0
Whether long int objects have calculations free of rounding errors: 1
Whether unsigned char objects have calculations free of rounding errors: 1

```

numeric_limits::is_iec559

Tests if a type conforms to IEC 559 standards.

```
static constexpr bool is_iec559 = false;
```

Return Value

true if the type conforms to the IEC 559 standards; **false** if not.

Remarks

The IEC 559 is an international standard for representing floating-point values and is also known as IEEE 754 in the USA.

Example

```
// numeric_limits_is_iec559.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects conform to iec559 standards: "
          << numeric_limits<float>::is_iec559
          << endl;
    cout << "Whether double objects conform to iec559 standards: "
          << numeric_limits<double>::is_iec559
          << endl;
    cout << "Whether int objects conform to iec559 standards: "
          << numeric_limits<int>::is_iec559
          << endl;
    cout << "Whether unsigned char objects conform to iec559 standards: "
          << numeric_limits<unsigned char>::is_iec559
          << endl;
}
```

```
Whether float objects conform to iec559 standards: 1
Whether double objects conform to iec559 standards: 1
Whether int objects conform to iec559 standards: 0
Whether unsigned char objects conform to iec559 standards: 0
```

numeric_limits::is_integer

Tests if a type has an integer representation.

```
static constexpr bool is_integer = false;
```

Return Value

true if the type has an integer representation; **false** if not.

Remarks

All predefined integer types have an integer representation.

Example

```

// numeric_limits_is_integer.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have an integral representation: "
          << numeric_limits<float>::is_integer
          << endl;
    cout << "Whether double objects have an integral representation: "
          << numeric_limits<double>::is_integer
          << endl;
    cout << "Whether int objects have an integral representation: "
          << numeric_limits<int>::is_integer
          << endl;
    cout << "Whether unsigned char objects have an integral representation: "
          << numeric_limits<unsigned char>::is_integer
          << endl;
}

```

```

Whether float objects have an integral representation: 0
Whether double objects have an integral representation: 0
Whether int objects have an integral representation: 1
Whether unsigned char objects have an integral representation: 1

```

numeric_limits::is_modulo

Tests if a **type** has a modulo representation.

```
static constexpr bool is_modulo = false;
```

Return Value

true if the type has a modulo representation; **false** if not.

Remarks

A modulo representation is a representation where all results are reduced modulo some value. All predefined unsigned integer types have a modulo representation.

Example

```

// numeric_limits_is_modulo.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have a modulo representation: "
          << numeric_limits<float>::is_modulo
          << endl;
    cout << "Whether double objects have a modulo representation: "
          << numeric_limits<double>::is_modulo
          << endl;
    cout << "Whether signed char objects have a modulo representation: "
          << numeric_limits<signed char>::is_modulo
          << endl;
    cout << "Whether unsigned char objects have a modulo representation: "
          << numeric_limits<unsigned char>::is_modulo
          << endl;
}

```

```

Whether float objects have a modulo representation: 0
Whether double objects have a modulo representation: 0
Whether signed char objects have a modulo representation: 1
Whether unsigned char objects have a modulo representation: 1

```

numeric_limits::is_signed

Tests if a type has a signed representation.

```
static constexpr bool is_signed = false;
```

Return Value

true if the type has a signed representation; **false** if not.

Remarks

The member stores true for a type that has a signed representation, which is the case for all predefined floating-point and signed integer types.

Example

```
// numeric_limits_is_signaled.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have a signed representation: "
          << numeric_limits<float>::is_signed
          << endl;
    cout << "Whether double objects have a signed representation: "
          << numeric_limits<double>::is_signed
          << endl;
    cout << "Whether signed char objects have a signed representation: "
          << numeric_limits<signed char>::is_signed
          << endl;
    cout << "Whether unsigned char objects have a signed representation: "
          << numeric_limits<unsigned char>::is_signed
          << endl;
}
```

```
Whether float objects have a signed representation: 1
Whether double objects have a signed representation: 1
Whether signed char objects have a signed representation: 1
Whether unsigned char objects have a signed representation: 0
```

numeric_limits::is_specialized

Tests if a type has an explicit specialization defined in the template class `numeric_limits`.

```
static constexpr bool is_specialized = false;
```

Return Value

true if the type has an explicit specialization defined in the template class; **false** if not.

Remarks

All scalar types other than pointers have an explicit specialization defined for template class `numeric_limits`.

Example


```
// numeric_limits_is_specialized.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float objects have an explicit "
          << "specialization in the class: "
          << numeric_limits<float>::is_specialized
          << endl;
    cout << "Whether float* objects have an explicit "
          << "specialization in the class: "
          << numeric_limits<float*>::is_specialized
          << endl;
    cout << "Whether int objects have an explicit "
          << "specialization in the class: "
          << numeric_limits<int>::is_specialized
          << endl;
    cout << "Whether int* objects have an explicit "
          << "specialization in the class: "
          << numeric_limits<int*>::is_specialized
          << endl;
}
```

```
Whether float objects have an explicit specialization in the class: 1
Whether float* objects have an explicit specialization in the class: 0
Whether int objects have an explicit specialization in the class: 1
Whether int* objects have an explicit specialization in the class: 0
```

numeric_limits::lowest

Returns the most negative finite value.

```
static constexpr Type lowest() throw();
```

Return Value

Returns the most negative finite value.

Remarks

Returns the most negative finite value for the type (which is typically `min()` for integer types and `-max()` for floating-point types). The return value is meaningful if `is_bounded` is **true**.

numeric_limits::max

Returns the maximum finite value for a type.

```
static constexpr Type max() throw();
```

Return Value

The maximum finite value for a type.

Remarks

The maximum finite value is `INT_MAX` for type **int** and `FLT_MAX` for type **float**. The return value is meaningful if

`is_bounded` is **true**.

Example

```
// numeric_limits_max.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main() {
    cout << "The maximum value for type float is: "
          << numeric_limits<float>::max( )
          << endl;
    cout << "The maximum value for type double is: "
          << numeric_limits<double>::max( )
          << endl;
    cout << "The maximum value for type int is: "
          << numeric_limits<int>::max( )
          << endl;
    cout << "The maximum value for type short int is: "
          << numeric_limits<short int>::max( )
          << endl;
}
```

numeric_limits::max_digits10

Returns the number of decimal digits required to make sure that two distinct values of the type have distinct decimal representations.

```
static constexpr int max_digits10 = 0;
```

Return Value

Returns the number of decimal digits that are required to make sure that two distinct values of the type have distinct decimal representations.

Remarks

The member stores the number of decimal digits required to make sure that two distinct values of the type have distinct decimal representations.

numeric_limits::max_exponent

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

```
static constexpr int max_exponent = 0;
```

Return Value

The maximum integral radix-based exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The `max_exponent` is the value `FLT_MAX_EXP` for type **float**.

Example

```

// numeric_limits_max_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The maximum radix-based exponent for type float is: "
          << numeric_limits<float>::max_exponent
          << endl;
    cout << "The maximum radix-based exponent for type double is: "
          << numeric_limits<double>::max_exponent
          << endl;
    cout << "The maximum radix-based exponent for type long double is: "
          << numeric_limits<long double>::max_exponent
          << endl;
}

```

```

The maximum radix-based exponent for type float is: 128
The maximum radix-based exponent for type double is: 1024
The maximum radix-based exponent for type long double is: 1024

```

numeric_limits::max_exponent10

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

```
static constexpr int max_exponent10 = 0;
```

Return Value

The maximum integral base 10 exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The `max_exponent` is the value `FLT_MAX_10` for type **float**.

Example

```
// numeric_limits_max_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The maximum base 10 exponent for type float is: "
          << numeric_limits<float>::max_exponent10
          << endl;
    cout << "The maximum base 10 exponent for type double is: "
          << numeric_limits<double>::max_exponent10
          << endl;
    cout << "The maximum base 10 exponent for type long double is: "
          << numeric_limits<long double>::max_exponent10
          << endl;
}
```

```
The maximum base 10 exponent for type float is: 38
The maximum base 10 exponent for type double is: 308
The maximum base 10 exponent for type long double is: 308
```

numeric_limits::min

Returns the minimum normalized value for a type.

```
static constexpr Type min() throw();
```

Return Value

The minimum normalized value for the type.

Remarks

The minimum normalized value is INT_MIN for type **int** and FLT_MIN for type **float**. The return value is meaningful if [is_bounded](#) is **true** or if [is_signed](#) is **false**.

Example

```

// numeric_limits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The minimum value for type float is: "
          << numeric_limits<float>::min( )
          << endl;
    cout << "The minimum value for type double is: "
          << numeric_limits<double>::min( )
          << endl;
    cout << "The minimum value for type int is: "
          << numeric_limits<int>::min( )
          << endl;
    cout << "The minimum value for type short int is: "
          << numeric_limits<short int>::min( )
          << endl;
}

```

```

The minimum value for type float is:  1.17549e-038
The minimum value for type double is:  2.22507e-308
The minimum value for type int is:    -2147483648
The minimum value for type short int is: -32768

```

numeric_limits::min_exponent

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

```
static constexpr int min_exponent = 0;
```

Return Value

The minimum integral radix-based exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The `min_exponent` is the value `FLT_MIN_EXP` for type **float**.

Example

```

// numeric_limits_min_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The minimum radix-based exponent for type float is:  "
          << numeric_limits<float>::min_exponent
          << endl;
    cout << "The minimum radix-based exponent for type double is:  "
          << numeric_limits<double>::min_exponent
          << endl;
    cout << "The minimum radix-based exponent for type long double is:  "
          << numeric_limits<long double>::min_exponent
          << endl;
}

```

```

The minimum radix-based exponent for type float is:  -125
The minimum radix-based exponent for type double is:  -1021
The minimum radix-based exponent for type long double is:  -1021

```

numeric_limits::min_exponent10

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

```
static constexpr int min_exponent10 = 0;
```

Return Value

The minimum integral base 10 exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The `min_exponent10` is the value `FLT_MIN_10_EXP` for type **float**.

Example

```

// numeric_limits_min_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The minimum base 10 exponent for type float is: "
          << numeric_limits<float>::min_exponent10
          << endl;
    cout << "The minimum base 10 exponent for type double is: "
          << numeric_limits<double>::min_exponent10
          << endl;
    cout << "The minimum base 10 exponent for type long double is: "
          << numeric_limits<long double>::min_exponent10
          << endl;
}

```

```

The minimum base 10 exponent for type float is: -37
The minimum base 10 exponent for type double is: -307
The minimum base 10 exponent for type long double is: -307

```

numeric_limits::quiet_NaN

Returns the representation of a quiet not a number (NaN) for the type.

```
static constexpr Type quiet_NaN() throw();
```

Return Value

The representation of a quiet NaN for the type.

Remarks

The return value is meaningful only if [has_quiet_NaN](#) is **true**.

Example

```

// numeric_limits_quiet_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The quiet NaN for type float is: "
          << numeric_limits<float>::quiet_NaN( )
          << endl;
    cout << "The quiet NaN for type int is: "
          << numeric_limits<int>::quiet_NaN( )
          << endl;
    cout << "The quiet NaN for type long double is: "
          << numeric_limits<long double>::quiet_NaN( )
          << endl;
}

```

```
The quiet NaN for type float is:  1.#QNAN
The quiet NaN for type int  is:   0
The quiet NaN for type long double is: 1.#QNAN
```

numeric_limits::radix

Returns the integral base, referred to as radix, used for the representation of a type.

```
static constexpr int radix = 0;
```

Return Value

The integral base for the representation of the type.

Remarks

The base is 2 for the predefined integer types, and the base to which the exponent is raised, or FLT_RADIX, for the predefined floating-point types.

Example

```
// numeric_limits_radix.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The base for type float is:  "
          << numeric_limits<float>::radix
          << endl;
    cout << "The base for type int is:  "
          << numeric_limits<int>::radix
          << endl;
    cout << "The base for type long double is:  "
          << numeric_limits<long double>::radix
          << endl;
}
```

```
The base for type float is:  2
The base for type int  is:   2
The base for type long double is:  2
```

numeric_limits::round_error

Returns the maximum rounding error for the type.

```
static constexpr Type round_error() throw();
```

Return Value

The maximum rounding error for the type.

Example


```
// numeric_limits_round_error.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The maximum rounding error for type float is: "
          << numeric_limits<float>::round_error( )
          << endl;
    cout << "The maximum rounding error for type int is: "
          << numeric_limits<int>::round_error( )
          << endl;
    cout << "The maximum rounding error for type long double is: "
          << numeric_limits<long double>::round_error( )
          << endl;
}
```

```
The maximum rounding error for type float is: 0.5
The maximum rounding error for type int is: 0
The maximum rounding error for type long double is: 0.5
```

numeric_limits::round_style

Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

```
static constexpr float_round_style round_style = round_toward_zero;
```

Return Value

A value from the `float_round_style` enumeration that describes the rounding style.

Remarks

The member stores a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

The round style is hard coded in this implementation, so even if the program starts up with a different rounding mode, that value will not change.

Example

```
// numeric_limits_round_style.cpp
// compile with: /EHsc
#include <iostream>
#include <float.h>
#include <limits>

using namespace std;

int main( )
{
    cout << "The rounding style for a double type is: "
          << numeric_limits<double>::round_style << endl;
    _controlfp_s(NULL, _RC_DOWN, _MCW_RC );
    cout << "The rounding style for a double type is now: "
          << numeric_limits<double>::round_style << endl;
    cout << "The rounding style for an int type is: "
          << numeric_limits<int>::round_style << endl;
}
```

```
The rounding style for a double type is: 1
The rounding style for a double type is now: 1
The rounding style for an int type is: 0
```

numeric_limits::signaling_NaN

Returns the representation of a signaling not a number (NaN) for the type.

```
static constexpr Type signaling_NaN() throw();
```

Return Value

The representation of a signaling NaN for the type.

Remarks

The return value is meaningful only if [has_signaling_NaN](#) is **true**.

Example

```
// numeric_limits_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "The signaling NaN for type float is: "
          << numeric_limits<float>::signaling_NaN( )
          << endl;
    cout << "The signaling NaN for type int is: "
          << numeric_limits<int>::signaling_NaN( )
          << endl;
    cout << "The signaling NaN for type long double is: "
          << numeric_limits<long double>::signaling_NaN( )
          << endl;
}
```

numeric_limits::tinyness_before

Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.

```
static constexpr bool tinyness_before = false;
```

Return Value

true if the type can detect tiny values before rounding; **false** if it cannot.

Remarks

Types that can detect tinyness were included as an option with IEC 559 floating-point representations and its implementation can affect some results.

Example

```
// numeric_limits_tinyness_before.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float types can detect tinyness before rounding: "
          << numeric_limits<float>::tinyness_before
          << endl;
    cout << "Whether double types can detect tinyness before rounding: "
          << numeric_limits<double>::tinyness_before
          << endl;
    cout << "Whether long int types can detect tinyness before rounding: "
          << numeric_limits<long int>::tinyness_before
          << endl;
    cout << "Whether unsigned char types can detect tinyness before rounding: "
          << numeric_limits<unsigned char>::tinyness_before
          << endl;
}
```

```
Whether float types can detect tinyness before rounding: 1
Whether double types can detect tinyness before rounding: 1
Whether long int types can detect tinyness before rounding: 0
Whether unsigned char types can detect tinyness before rounding: 0
```

numeric_limits::traps

Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

```
static constexpr bool traps = false;
```

Return Value

true if trapping is implemented for the type; **false** if it is not.

Example

```
// numeric_limits_traps.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
    cout << "Whether float types have implemented trapping: "
          << numeric_limits<float>::traps
          << endl;
    cout << "Whether double types have implemented trapping: "
          << numeric_limits<double>::traps
          << endl;
    cout << "Whether long int types have implemented trapping: "
          << numeric_limits<long int>::traps
          << endl;
    cout << "Whether unsigned char types have implemented trapping: "
          << numeric_limits<unsigned char>::traps
          << endl;
}
```

```
Whether float types have implemented trapping: 1
Whether double types have implemented trapping: 1
Whether long int types have implemented trapping: 0
Whether unsigned char types have implemented trapping: 0
```

See also

[Thread Safety in the C++ Standard Library](#)

<list>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class list and several supporting templates.

Syntax

```
#include <list>
```

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the list object on the left side of the operator is not equal to the list object on the right side.
<code>operator<</code>	Tests if the list object on the left side of the operator is less than the list object on the right side.
<code>operator<=</code>	Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.
<code>operator==</code>	Tests if the list object on the left side of the operator is equal to the list object on the right side.
<code>operator></code>	Tests if the list object on the left side of the operator is greater than the list object on the right side.
<code>operator>=</code>	Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

Classes

CLASS	DESCRIPTION
<code>list Class</code>	A template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<list> operators

10/31/2018 • 5 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Tests if the list object on the left side of the operator is not equal to the list object on the right side.

```
bool operator!=(  
    const list<Type, Allocator>& left,  
    const list<Type, Allocator>& right);
```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the lists are not equal; **false** if the lists are equal.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// list_op_ne.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2;
    c1.push_back( 1 );
    c2.push_back( 2 );

    if ( c1 != c2 )
        cout << "Lists not equal." << endl;
    else
        cout << "Lists equal." << endl;
}
/* Output:
Lists not equal.
*/

```

operator<

Tests if the list object on the left side of the operator is less than the list object on the right side.

```

bool operator<(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);

```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// list_op_lt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2;
    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 4 );

    c2.push_back( 1 );
    c2.push_back( 3 );

    if ( c1 < c2 )
        cout << "List c1 is less than list c2." << endl;
    else
        cout << "List c1 is not less than list c2." << endl;
}
/* Output:
List c1 is less than list c2.
*/

```

operator<=

Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.

```

bool operator<=(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);

```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example


```

// list_op_le.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2;
    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 4 );

    c2.push_back( 1 );
    c2.push_back( 3 );

    if ( c1 <= c2 )
        cout << "List c1 is less than or equal to list c2." << endl;
    else
        cout << "List c1 is greater than list c2." << endl;
}
/* Output:
List c1 is less than or equal to list c2.
*/

```

operator==

Tests if the list object on the left side of the operator is equal to the list object on the right side.

```

bool operator==(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);

```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the list on the left side of the operator is equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// list_op_eq.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
    using namespace std;

    list<int> c1, c2;
    c1.push_back( 1 );
    c2.push_back( 1 );

    if ( c1 == c2 )
        cout << "The lists are equal." << endl;
    else
        cout << "The lists are not equal." << endl;
}
/* Output:
The lists are equal.
*/

```

operator>

Tests if the list object on the left side of the operator is greater than the list object on the right side.

```

bool operator>(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);

```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// list_op_gt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
    using namespace std;
    list<int> c1, c2;
    c1.push_back( 1 );
    c1.push_back( 3 );
    c1.push_back( 1 );

    c2.push_back( 1 );
    c2.push_back( 2 );
    c2.push_back( 2 );

    if ( c1 > c2 )
        cout << "List c1 is greater than list c2." << endl;
    else
        cout << "List c1 is not greater than list c2." << endl;
}
/* Output:
List c1 is greater than list c2.
*/

```

operator>=

Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

```

bool operator>=(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);

```

Parameters

left

An object of type `list`.

right

An object of type `list`.

Return Value

true if the list on the left side of the operator is greater than or equal to the list on the right side of the operator; otherwise **false**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// list_op_ge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2;
    c1.push_back( 1 );
    c1.push_back( 3 );
    c1.push_back( 1 );

    c2.push_back( 1 );
    c2.push_back( 2 );
    c2.push_back( 2 );

    if ( c1 >= c2 )
        cout << "List c1 is greater than or equal to list c2." << endl;
    else
        cout << "List c1 is less than list c2." << endl;
}
/* Output:
List c1 is greater than or equal to list c2.
*/

```

See also

[<list>](#)

list Class

11/9/2018 • 45 minutes to read • [Edit Online](#)

The C++ Standard Library list class is a template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type *Type*.

Syntax

```
template <class Type, class Allocator= allocator<Type>>
class list
```

Parameters

Type

The element data type to be stored in the list.

Allocator

The type that represents the stored allocator object that encapsulates details about the list's allocation and deallocation of memory. This argument is optional, and the default value is **allocator<Type>**.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the class deque container is superior when random access is needed and insertions and deletions at both the beginning and the end of a sequence are at a premium.

The list member functions [merge](#), [reverse](#), [unique](#), [remove](#), and [remove_if](#) have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid.

Include the C++ Standard Library standard header `<list>` to define the [container](#) template class list and several supporting templates.

Constructors

CONSTRUCTOR	DESCRIPTION
list	Constructs a list of a specific size or with elements of a specific value or with a specific <code>allocator</code> or as a copy of some other list.

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	A type that represents the <code>allocator</code> class for a list object.

TYPE NAME	DESCRIPTION
<code>const_iterator</code>	A type that provides a bidirectional iterator that can read a const element in a list.
<code>const_pointer</code>	A type that provides a pointer to a const element in a list.
<code>const_reference</code>	A type that provides a reference to a const element stored in a list for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in a list.
<code>difference_type</code>	A type that provides the difference between two iterators that refer to elements within the same list.
<code>iterator</code>	A type that provides a bidirectional iterator that can read or modify any element in a list.
<code>pointer</code>	A type that provides a pointer to an element in a list.
<code>reference</code>	A type that provides a reference to a const element stored in a list for reading and performing const operations.
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed list.
<code>size_type</code>	A type that counts the number of elements in a list.
<code>value_type</code>	A type that represents the data type stored in a list.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Erases elements from a list and copies a new set of elements to the target list.
<code>back</code>	Returns a reference to the last element of a list.
<code>begin</code>	Returns an iterator addressing the first element in a list.
<code>cbegin</code>	Returns a const iterator addressing the first element in a list.
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a list.
<code>clear</code>	Erases all the elements of a list.
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed list.
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed list.

MEMBER FUNCTION	DESCRIPTION
<code>emplace</code>	Inserts an element constructed in place into a list at a specified position.
<code>emplace_back</code>	Adds an element constructed in place to the end of a list.
<code>emplace_front</code>	Adds an element constructed in place to the beginning of a list.
<code>empty</code>	Tests if a list is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a list.
<code>erase</code>	Removes an element or a range of elements in a list from specified positions.
<code>front</code>	Returns a reference to the first element in a list.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct a list.
<code>insert</code>	Inserts an element or a number of elements or a range of elements into a list at a specified position.
<code>max_size</code>	Returns the maximum length of a list.
<code>merge</code>	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
<code>pop_back</code>	Deletes the element at the end of a list.
<code>pop_front</code>	Deletes the element at the beginning of a list.
<code>push_back</code>	Adds an element to the end of a list.
<code>push_front</code>	Adds an element to the beginning of a list.
<code>rbegin</code>	Returns an iterator addressing the first element in a reversed list.
<code>remove</code>	Erases elements in a list that match a specified value.
<code>remove_if</code>	Erases elements from the list for which a specified predicate is satisfied.
<code>rend</code>	Returns an iterator that addresses the location succeeding the last element in a reversed list.
<code>resize</code>	Specifies a new size for a list.
<code>reverse</code>	Reverses the order in which the elements occur in a list.

MEMBER FUNCTION	DESCRIPTION
size	Returns the number of elements in a list.
sort	Arranges the elements of a list in ascending order or with respect to some other order relation.
splice	Removes elements from the argument list and inserts them into the target list.
swap	Exchanges the elements of two lists.
unique	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.

Operators

OPERATOR	DESCRIPTION
list::operator=	Replaces the elements of the list with a copy of another list.

Requirements

Header: <list>

list::allocator_type

A type that represents the allocator class for a list object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter *Allocator*.

Example

See the example for [get_allocator](#).

list::assign

Erases elements from a list and copies a new set of elements to a target list.

```
void assign(
    size_type Count,
    const Type& Val);

void assign
    initializer_list<Type> IList);

template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);
```

Parameters

First

Position of the first element in the range of elements to be copied from the argument list.

Last

Position of the first element just beyond the range of elements to be copied from the argument list.

Count

The number of copies of an element being inserted into the list.

Val

The value of the element being inserted into the list.

lList

The initializer_list that contains the elements to be inserted.

Remarks

After erasing any existing elements in the target list, assign either inserts a specified range of elements from the original list or from some other list into the target list or inserts copies of a new element of a specified value into the target list

Example

```
// list_assign.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    list<int> c1, c2;
    list<int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(7, 4);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign({ 10, 20, 30, 40 });
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;
}
```

```
c1 = 10 20 30 c1 = 50 60 c1 = 4 4 4 4 4 4 c1 = 10 20 30 40
```

list::back

Returns a reference to the last element of a list.

```
reference back();

const_reference back() const;
```

Return Value

The last element of the list. If the list is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the list object cannot be modified. If the return value of `back` is assigned to a `reference`, the list object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty list. See [Checked Iterators](#) for more information.

Example

```
// list_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );
    c1.push_back( 11 );

    int& i = c1.back( );
    const int& ii = c1.front( );

    cout << "The last integer of c1 is " << i << endl;
    i--;
    cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

list::begin

Returns an iterator addressing the first element in a list.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the list or to the location succeeding an empty list.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the list object cannot be modified.

If the return value of `begin` is assigned to an `iterator`, the elements in the list object can be modified.

Example

```
// list_begin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;
    list<int>::const_iterator c1_cIter;

    c1.push_back( 1 );
    c1.push_back( 2 );

    c1_Iter = c1.begin( );
    cout << "The first element of c1 is " << *c1_Iter << endl;

    *c1_Iter = 20;
    c1_Iter = c1.begin( );
    cout << "The first element of c1 is now " << *c1_Iter << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}
```

```
The first element of c1 is 1
The first element of c1 is now 20
```

list::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

list::cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

list::clear

Erases all the elements of a list.

```
void clear();
```

Example

```
// list_clear.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    cout << "The size of the list is initially " << c1.size( ) << endl;
    c1.clear( );
    cout << "The size of list after clearing is " << c1.size( ) << endl;
}
```

```
The size of the list is initially 3
The size of list after clearing is 0
```

list::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in a list.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for [back](#).

list::const_pointer

Provides a pointer to a **const** element in a list.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a list object.

list::const_reference

A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

Example

```
// list_const_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );

    const list<int> c2 = c1;
    const int &i = c2.front( );
    const int &j = c2.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

    // The following line would cause an error because c2 is const
    // c2.push_back( 30 );
}
```

```
The first element is 10
The second element is 20
```

list::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in a list.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the list in reverse.

Example

See the example for [rbegin](#).

list::crbegin

Returns a const iterator addressing the first element in a reversed list.

```
const_reverse_iterator rbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed `list`).

Remarks

`crbegin` is used with a reversed list just as [list::begin](#) is used with a `list`.

With the return value of `crbegin`, the list object cannot be modified. [list::rbegin](#) can be used to iterate through a list backwards.

Example

```
// list_crbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::const_reverse_iterator c1_crIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1_crIter = c1.crbegin( );
    cout << "The last element in the list is " << *c1_crIter << "." << endl;
}
```

The last element in the list is 30.

list::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed list.

```
const_reverse_iterator rend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [list](#) (the location that had preceded the first element in the unreversed `list`).

Remarks

`crend` is used with a reversed list just as [list::end](#) is used with a `list`.

With the return value of `crend`, the `list` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `list`.

The value returned by `crend` should not be dereferenced.

Example

```

// list_crend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::const_reverse_iterator c1_crIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_crIter = c1.crend( );
    c1_crIter --; // Decrementing a reverse iterator moves it forward in
                  // the list (to point to the first element here)
    cout << "The first element in the list is: " << *c1_crIter << endl;
}

```

```
The first element in the list is: 10
```

list::difference_type

A signed integer type that can be used to represent the number of elements of a list in a range between elements pointed to by iterators.

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container.

The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like `set`, subtraction between iterators is only supported by random-access iterators provided by a random-access container, such as [vector Class](#).

Example


```

// list_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <list>
#include <algorithm>

int main( )
{
    using namespace std;

    list<int> c1;
    list<int>::iterator c1_Iter, c2_Iter;

    c1.push_back( 30 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1.push_back( 10 );
    c1.push_back( 30 );
    c1.push_back( 20 );

    c1_Iter = c1.begin( );
    c2_Iter = c1.end( );

    list<int>::difference_type df_typ1, df_typ2, df_typ3;

    df_typ1 = count( c1_Iter, c2_Iter, 10 );
    df_typ2 = count( c1_Iter, c2_Iter, 20 );
    df_typ3 = count( c1_Iter, c2_Iter, 30 );
    cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
    cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
    cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}

```

```

The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.

```

list::emplace

Inserts an element constructed in place into a list at a specified position.

```
void emplace(iterator Where, Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>Where</i>	The position in the target list where the first element is inserted.
<i>val</i>	The element added to the end of the <code>list</code> .

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

```
// list_emplace.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    list<string> c2;
    string str("a");

    c2.emplace(c2.begin(), move( str ) );
    cout << "Moved first element: " << c2.back( ) << endl;
}
```

Moved first element: a

list::emplace_back

Adds an element constructed in place to the end of a list.

```
void emplace_back(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the end of the list .

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

```
// list_emplace_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    list<string> c2;
    string str("a");

    c2.emplace_back( move( str ) );
    cout << "Moved first element: " << c2.back( ) << endl;
}
```

Moved first element: a

list::emplace_front

Adds an element constructed in place to the beginning of a list.

```
void emplace_front(Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the list .

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

```
// list_emplace_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    list <string> c2;
    string str("a");

    c2.emplace_front( move( str ) );
    cout << "Moved first element: " << c2.front( ) << endl;
}
```

```
Moved first element: a
```

list::empty

Tests if a list is empty.

```
bool empty() const;
```

Return Value

true if the list is empty; **false** if the list is not empty.

Example

```

// list_empty.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );
    if ( c1.empty( ) )
        cout << "The list is empty." << endl;
    else
        cout << "The list is not empty." << endl;
}

```

The list is not empty.

list::end

Returns an iterator that addresses the location succeeding the last element in a list.

```

const_iterator end() const;
iterator end();

```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a list. If the list is empty, then

```
list::end == list::begin .
```

Remarks

`end` is used to test whether an iterator has reached the end of its list.

Example

```

// list_end.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_Iter = c1.end( );
    c1_Iter--;
    cout << "The last integer of c1 is " << *c1_Iter << endl;

    c1_Iter--;
    *c1_Iter = 400;
    cout << "The new next-to-last integer of c1 is "
        << *c1_Iter << endl;

    // If a const iterator had been declared instead with the line:
    // list<int>::const_iterator c1_Iter;
    // an error would have resulted when inserting the 400

    cout << "The list is now:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
}

```

```

The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The list is now: 10 400 30

```

list::erase

Removes an element or a range of elements in a list from specified positions.

```

iterator erase(iterator Where);
iterator erase(iterator first, iterator last);

```

Parameters

Where

Position of the element to be removed from the list.

first

Position of the first element removed from the list.

last

Position just beyond the last element removed from the list.

Return Value

A bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the list if no such element exists.

Remarks

No reallocation occurs, so iterators and references become invalid only for the erased elements.

`erase` never throws an exception.

Example

```
// list_erase.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1.push_back( 40 );
    c1.push_back( 50 );
    cout << "The initial list is:";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << " " << *Iter;
    cout << endl;

    c1.erase( c1.begin( ) );
    cout << "After erasing the first element, the list becomes:";
    for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << " " << *Iter;
    cout << endl;
    Iter = c1.begin( );
    Iter++;
    c1.erase( Iter, c1.end( ) );
    cout << "After erasing all elements but the first, the list becomes: ";
    for (Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
        cout << " " << *Iter;
    cout << endl;
}
```

```
The initial list is: 10 20 30 40 50
After erasing the first element, the list becomes: 20 30 40 50
After erasing all elements but the first, the list becomes:  20
```

list::front

Returns a reference to the first element in a list.

```
reference front();
const_reference front() const;
```

Return Value

If the list is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the list object cannot be modified. If the return value of `front` is assigned to a `reference`, the list object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you

attempt to access an element in an empty list. See [Checked Iterators](#) for more information.

Example

```
// list_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );

    int& i = c1.front();
    const int& ii = c1.front();

    cout << "The first integer of c1 is " << i << endl;
    i++;
    cout << "The first integer of c1 is " << ii << endl;
}
```

```
The first integer of c1 is 10
The first integer of c1 is 11
```

list::get_allocator

Returns a copy of the allocator object used to construct a list.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the list.

Remarks

Allocators for the list class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// list_get_allocator.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    // The following lines declare objects
    // that use the default allocator.
    list<int> c1;
    list<int, allocator<int> > c2 = list<int, allocator<int> >( allocator<int>( ) );

    // c3 will use the same allocator class as c1
    list<int> c3( c1.get_allocator( ) );

    list<int>::allocator_type xlst = c1.get_allocator( );
    // You can now call functions on the allocator class used by c1
}

```

list::insert

Inserts an element or a number of elements or a range of elements into a list at a specified position.

```

iterator insert(iterator Where, const Type& Val);
iterator insert(iterator Where, Type&& Val);

void insert(iterator Where, size_type Count, const Type& Val);
iterator insert(iterator Where, initializer_list<Type> Ilist);

template <class InputIterator>
void insert(iterator Where, InputIterator First, InputIterator Last);

```

Parameters

PARAMETER	DESCRIPTION
<i>Where</i>	The position in the target list where the first element is inserted.
<i>Val</i>	The value of the element being inserted into the list.
<i>Count</i>	The number of elements being inserted into the list.
<i>First</i>	The position of the first element in the range of elements in the argument list to be copied.
<i>Last</i>	The position of the first element beyond the range of elements in the argument list to be copied.

Return Value

The first two insert functions return an iterator that points to the position where the new element was inserted into the list.

Example


```

// list_class_insert.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    list<int> c1, c2;
    list<int>::iterator Iter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    c1.insert(Iter, 100);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    Iter++;
    c1.insert(Iter, 2, 200);

    cout << "c1 =";
    for(auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.insert(++c1.begin(), c2.begin(), --c2.end());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    // initialize a list of strings by moving
    list< string > c3;
    string str("a");

    c3.insert(c3.begin(), move(str));
    cout << "Moved first element: " << c3.front() << endl;

    // Assign with an initializer_list
    list<int> c4{ {1, 2, 3, 4} };
    c4.insert(c4.begin(), { 5, 6, 7, 8 });

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;
}

```

list::iterator

A type that provides a bidirectional iterator that can read or modify any element in a list.

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for [begin](#).

list::list

Constructs a list of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other list.

```
list();
explicit list(const Allocator& Al);
explicit list(size_type Count);
list(size_type Count, const Type& Val);
list(size_type Count, const Type& Val, const Allocator& Al);

list(const list& Right);
list(list&& Right);
list(initializer_list<Type> IList, const Allocator& Al);

template <class InputIterator>
list(InputIterator First, InputIterator Last);

template <class InputIterator>
list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The allocator class to use with this object.
<i>Count</i>	The number of elements in the list constructed.
<i>Val</i>	The value of the elements in the list.
<i>Right</i>	The list of which the constructed list is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.
<i>IList</i>	The initializer_list that contains the elements to be copied.

Remarks

All constructors store an allocator object (*Al*) and initialize the list.

`get_allocator` returns a copy of the allocator object used to construct a list.

The first two constructors specify an empty initial list, the second specifying the allocator type (*Al*) to be used.

The third constructor specifies a repetition of a specified number (*Count*) of elements of the default value for class `Type`.

The fourth and fifth constructors specify a repetition of (*Count*) elements of value *Val*.

The sixth constructor specifies a copy of the list *Right*.

The seventh constructor moves the list *Right*.

The eighth constructor uses an `initializer_list` to specify the elements.

The next two constructors copy the range `[First, Last)` of a list.

None of the constructors perform any interim reallocations.

Example

```
// list_class_list.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    // Create an empty list c0
    list<int> c0;

    // Create a list c1 with 3 elements of default value 0
    list<int> c1(3);

    // Create a list c2 with 5 elements of value 2
    list<int> c2(5, 2);

    // Create a list c3 with 3 elements of value 1 and with the
    // allocator of list c2
    list<int> c3(3, 1, c2.get_allocator());

    // Create a copy, list c4, of list c2
    list<int> c4(c2);

    // Create a list c5 by copying the range c4[ first, last)
    list<int>::iterator c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    list<int> c5(c4.begin(), c4_Iter);

    // Create a list c6 by copying the range c4[ first, last) and with
    // the allocator of list c2
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    c4_Iter++;
    list<int> c6(c4.begin(), c4_Iter, c2.get_allocator());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    cout << "c2 =";
    for (auto c : c2)
        cout << " " << c;
```

```

    cout << endl;

    cout << "c3 =";
    for (auto c : c3)
        cout << " " << c;
    cout << endl;

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;

    cout << "c5 =";
    for (auto c : c5)
        cout << " " << c;
    cout << endl;

    cout << "c6 =";
    for (auto c : c6)
        cout << " " << c;
    cout << endl;

    // Move list c6 to list c7
    list<int> c7(move(c6));
    cout << "c7 =";
    for (auto c : c7)
        cout << " " << c;
    cout << endl;

    // Construct with initializer_list
    list<int> c8({ 1, 2, 3, 4 });
    cout << "c8 =";
    for (auto c : c8)
        cout << " " << c;
    cout << endl;
}

```

c1 = 0 0 0 c2 = 2 2 2 2 2 c3 = 1 1 1 c4 = 2 2 2 2 2 c5 = 2 2 c6 = 2 2 2 c7 = 2 2 2 c8 = 1 2 3 4

list::max_size

Returns the maximum length of a list.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the list.

Example

```
// list_max_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::size_type i;

    i = c1.max_size( );
    cout << "Maximum possible length of the list is " << i << "." << endl;
}
```

list::merge

Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.

```
void merge(list<Type, Allocator>& right);

template <class Traits>
void merge(list<Type, Allocator>& right, Traits comp);
```

Parameters

right

The argument list to be merged with the target list.

comp

The comparison operator used to order the elements of the target list.

Remarks

The argument list *right* is merged with the target list.

Both argument and target lists must be ordered with the same comparison relation by which the resulting sequence is to be ordered. The default order for the first member function is ascending order. The second member function imposes the user-specified comparison operation *comp* of class `Traits`.

Example

```

// list_merge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2, c3;
    list<int>::iterator c1_Iter, c2_Iter, c3_Iter;

    c1.push_back( 3 );
    c1.push_back( 6 );
    c2.push_back( 2 );
    c2.push_back( 4 );
    c3.push_back( 5 );
    c3.push_back( 1 );

    cout << "c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    cout << "c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;

    c2.merge( c1 ); // Merge c1 into c2 in (default) ascending order
    c2.sort( greater<int>( ) );
    cout << "After merging c1 with c2 and sorting with >: c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;

    cout << "c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
        cout << " " << *c3_Iter;
    cout << endl;

    c2.merge( c3, greater<int>( ) );
    cout << "After merging c3 with c2 according to the '>' comparison relation: c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;
}

```

```

c1 = 3 6
c2 = 2 4
After merging c1 with c2 and sorting with >: c2 = 6 4 3 2
c3 = 5 1
After merging c3 with c2 according to the '>' comparison relation: c2 = 6 5 4 3 2 1

```

list::operator=

Replaces the elements of the list with a copy of another list.

```

list& operator=(const list& right);
list& operator=(list&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The list being copied into the <code>list</code> .

Remarks

After erasing any existing elements in a `list`, the operator either copies or moves the contents of *right* into the `list`.

Example

```
// list_operator_as.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> v1, v2, v3;
    list<int>::iterator iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = forward< list<int> >(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

list::pointer

Provides a pointer to an element in a list.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a list object.

list::pop_back

Deletes the element at the end of a list.

```
void pop_back();
```

Remarks

The last element must not be empty. `pop_back` never throws an exception.

Example

```
// list_pop_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 1 );
    c1.push_back( 2 );
    cout << "The first element is: " << c1.front( ) << endl;
    cout << "The last element is: " << c1.back( ) << endl;

    c1.pop_back( );
    cout << "After deleting the element at the end of the list, "
         << "the last element is: " << c1.back( ) << endl;
}
```

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the list, the last element is: 1
```

list::pop_front

Deletes the element at the beginning of a list.

```
void pop_front();
```

Remarks

The first element must not be empty. `pop_front` never throws an exception.

Example


```
// list_pop_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 1 );
    c1.push_back( 2 );
    cout << "The first element is: " << c1.front( ) << endl;
    cout << "The second element is: " << c1.back( ) << endl;

    c1.pop_front( );
    cout << "After deleting the element at the beginning of the list, "
         << "the first element is: " << c1.front( ) << endl;
}
```

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the list, the first element is: 2
```

list::push_back

Adds an element to the end of a list.

```
void push_back(const Type&& val);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the end of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

```

// list_push_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 1 );
    if ( c1.size( ) != 0 )
        cout << "Last element: " << c1.back( ) << endl;

    c1.push_back( 2 );
    if ( c1.size( ) != 0 )
        cout << "New last element: " << c1.back( ) << endl;

    // move initialize a list of strings
    list<string> c2;
    string str("a");

    c2.push_back( move( str ) );
    cout << "Moved first element: " << c2.back( ) << endl;
}

```

```

Last element: 1
New last element: 2
Moved first element: a

```

list::push_front

Adds an element to the beginning of a list.

```

void push_front(const Type& val);
void push_front(Type&& val);

```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The element added to the beginning of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

```

// list_push_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_front( 1 );
    if ( c1.size( ) != 0 )
        cout << "First element: " << c1.front( ) << endl;

    c1.push_front( 2 );
    if ( c1.size( ) != 0 )
        cout << "New first element: " << c1.front( ) << endl;

    // move initialize a list of strings
    list<string> c2;
    string str("a");

    c2.push_front( move( str ) );
    cout << "Moved first element: " << c2.front( ) << endl;
}

```

```

First element: 1
New first element: 2
Moved first element: a

```

list::rbegin

Returns an iterator that addresses the first element in a reversed list.

```

const_reverse_iterator rbegin() const;
reverse_iterator rbegin();

```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed list).

Remarks

`rbegin` is used with a reversed list just as `begin` is used with a list.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the list object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the list object can be modified.

`rbegin` can be used to iterate through a list backwards.

Example

```

// list_rbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;
    list<int>::reverse_iterator c1_rIter;

    // If the following line replaced the line above, *c1_rIter = 40;
    // (below) would be an error
    //list<int>::const_reverse_iterator c1_rIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1_rIter = c1.rbegin( );
    cout << "The last element in the list is " << *c1_rIter << "." << endl;

    cout << "The list is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    // rbegin can be used to start an iteration through a list in
    // reverse order
    cout << "The reversed list is:";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << " " << *c1_rIter;
    cout << endl;

    c1_rIter = c1.rbegin( );
    *c1_rIter = 40;
    cout << "The last element in the list is now " << *c1_rIter << "." << endl;
}

```

```

The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.

```

list::reference

A type that provides a reference to an element stored in a list.

```

typedef typename Allocator::reference reference;

```

Example

```
// list_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );

    int &i = c1.front( );
    int &j = c1.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;
}
```

```
The first element is 10
The second element is 20
```

list::remove

Erases elements in a list that match a specified value.

```
void remove(const Type& val);
```

Parameters

val

The value which, if held by an element, will result in that element's removal from the list.

Remarks

The order of the elements remaining is not affected.

Example

```

// list_remove.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter, c2_Iter;

    c1.push_back( 5 );
    c1.push_back( 100 );
    c1.push_back( 5 );
    c1.push_back( 200 );
    c1.push_back( 5 );
    c1.push_back( 300 );

    cout << "The initial list is c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    list<int> c2 = c1;
    c2.remove( 5 );
    cout << "After removing elements with value 5, the list becomes c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;
}

```

The initial list is c1 = 5 100 5 200 5 300
 After removing elements with value 5, the list becomes c2 = 100 200 300

list::remove_if

Erases elements from a list for which a specified predicate is satisfied.

```

template <class Predicate>
void remove_if(Predicate pred)

```

Parameters

pred

The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Example

```

// list_remove_if.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

template <class T> class is_odd : public std::unary_function<T, bool>
{
public:
    bool operator( ) ( T& val )
    {
        return ( val % 2 ) == 1;
    }
};

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter, c2_Iter;

    c1.push_back( 3 );
    c1.push_back( 4 );
    c1.push_back( 5 );
    c1.push_back( 6 );
    c1.push_back( 7 );
    c1.push_back( 8 );

    cout << "The initial list is c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    list<int> c2 = c1;
    c2.remove_if( is_odd<int>( ) );

    cout << "After removing the odd elements, "
        << "the list becomes c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;
}

```

```

The initial list is c1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes c2 = 4 6 8

```

list::rend

Returns an iterator that addresses the location that follows the last element in a reversed list.

```

const_reverse_iterator rend() const;
reverse_iterator rend();

```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed list).

Remarks

`rend` is used with a reversed list just as `end` is used with a list.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the list object cannot be modified. If the

return value of `rend` is assigned to a `reverse_iterator`, the list object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its list.

The value returned by `rend` should not be dereferenced.

Example

```
// list_rend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;
    list<int>::reverse_iterator c1_rIter;

    // If the following line had replaced the line above, an error would
    // have resulted in the line modifying an element (commented below)
    // because the iterator would have been const
    // list<int>::const_reverse_iterator c1_rIter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_rIter = c1.rend( );
    c1_rIter --; // Decrementing a reverse iterator moves it forward in
                // the list (to point to the first element here)
    cout << "The first element in the list is: " << *c1_rIter << endl;

    cout << "The list is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    // rend can be used to test if an iteration is through all of the
    // elements of a reversed list
    cout << "The reversed list is:";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << " " << *c1_rIter;
    cout << endl;

    c1_rIter = c1.rend( );
    c1_rIter--; // Decrementing the reverse iterator moves it backward
               // in the reversed list (to the last element here)

    *c1_rIter = 40; // This modification of the last element would have
                   // caused an error if a const_reverse iterator had
                   // been declared (as noted above)

    cout << "The modified reversed list is:";
    for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
        cout << " " << *c1_rIter;
    cout << endl;
}
```

```
The first element in the list is: 10
The list is: 10 20 30
The reversed list is: 30 20 10
The modified reversed list is: 30 20 40
```


list::resize

Specifies a new size for a list.

```
void resize(size_type _Nsize);  
void resize(size_type _Nsize, Type val);
```

Parameters

_Nsize

The new size of the list.

val

The value of the new elements to be added to the list if the new size is larger than the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the list's size is less than the requested size, *_Nsize*, elements are added to the list until it reaches the requested size.

If the list's size is larger than the requested size, the elements closest to the end of the list are deleted until the list reaches the size *_Nsize*.

If the present size of the list is the same as the requested size, no action is taken.

[size](#) reflects the current size of the list.

Example

```
// list_resize.cpp  
// compile with: /EHsc  
#include <list>  
#include <iostream>  
  
int main( )  
{  
    using namespace std;  
    list<int> c1;  
  
    c1.push_back( 10 );  
    c1.push_back( 20 );  
    c1.push_back( 30 );  
  
    c1.resize( 4,40 );  
    cout << "The size of c1 is " << c1.size( ) << endl;  
    cout << "The value of the last element is " << c1.back( ) << endl;  
  
    c1.resize( 5 );  
    cout << "The size of c1 is now " << c1.size( ) << endl;  
    cout << "The value of the last element is now " << c1.back( ) << endl;  
  
    c1.resize( 2 );  
    cout << "The reduced size of c1 is: " << c1.size( ) << endl;  
    cout << "The value of the last element is now " << c1.back( ) << endl;  
}
```

```
The size of c1 is 4
The value of the last element is 40
The size of c1 is now 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

list::reverse

Reverses the order in which the elements occur in a list.

```
void reverse();
```

Example

```
// list_reverse.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    cout << "c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    c1.reverse( );
    cout << "Reversed c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;
}
```

```
c1 = 10 20 30
Reversed c1 = 30 20 10
```

list::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the list in reverse.

Example

See the example for [rbegin](#).

list::size

Returns the number of elements in a list.

```
size_type size() const;
```

Return Value

The current length of the list.

Example

```
// list_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::size_type i;

    c1.push_back( 5 );
    i = c1.size( );
    cout << "List length is " << i << "." << endl;

    c1.push_back( 7 );
    i = c1.size( );
    cout << "List length is now " << i << "." << endl;
}
```

```
List length is 1.
List length is now 2.
```

list::size_type

A type that counts the number of elements in a list.

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for [size](#).

list::sort

Arranges the elements of a list in ascending order or with respect to some other user-specified order.

```
void sort();

template <class Traits>
void sort(Traits comp);
```

Parameters

comp

The comparison operator used to order successive elements.

Remarks

The first member function puts the elements in ascending order by default.

The member template function orders the elements according to the user-specified comparison operation *comp* of class `Traits`.

Example

```
// list_sort.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter;

    c1.push_back( 20 );
    c1.push_back( 10 );
    c1.push_back( 30 );

    cout << "Before sorting: c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    c1.sort( );
    cout << "After sorting c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    c1.sort( greater<int>( ) );
    cout << "After sorting with 'greater than' operation, c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;
}
```

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

list::splice

Removes elements from a source list and inserts them into a destination list.

```
// insert the entire source list
void splice(const_iterator Where, list<Type, Allocator>& Source);
void splice(const_iterator Where, list<Type, Allocator>&& Source);

// insert one element of the source list
void splice(const_iterator Where, list<Type, Allocator>& Source, const_iterator Iter);
void splice(const_iterator Where, list<Type, Allocator>&& Source, const_iterator Iter);

// insert a range of elements from the source list
void splice(const_iterator Where, list<Type, Allocator>& Source, const_iterator First, const_iterator Last);
void splice(const_iterator Where, list<Type, Allocator>&& Source, const_iterator First, const_iterator Last);
```

Parameters

Where

The position in the destination list before which to insert.

Source

The source list that is to be inserted into the destination list.

Iter

The element to be inserted from the source list.

First

The first element in the range to be inserted from the source list.

Last

The first position beyond the last element in the range to be inserted from the source list.

Remarks

The first pair of member functions inserts all elements in the source list into the destination list before the position referred to by *Where* and removes all elements from the source list. (`&Source` must not equal `this`.)

The second pair of member functions inserts the element referred to by *Iter* before the position in the destination list referred to by *Where* and removes *Iter* from the source list. (If `Where == Iter || Where == ++Iter`, no change occurs.)

The third pair of member functions inserts the range designated by [`First`, `Last`) before the element in the destination list referred to by *Where* and removes that range of elements from the source list. (If `&Source == this`, the range [`First`, `Last`) must not include the element pointed to by *Where*.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class [iterator](#) is incremented `N` times.

In all cases iterators, pointers, or references that refer to spliced elements remain valid and are transferred to the destination container.

Example

```
// list_splice.cpp
// compile with: /EHsc /W4
#include <list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    list<int> c1{10,11};
    list<int> c2{20,21,22};
    list<int> c3{30,31};
    list<int> c4{40,41,42,43};

    list<int>::iterator where_iter;
    list<int>::iterator first_iter;
```

```

list<int>::iterator last_iter;

cout << "Beginning state of lists:" << endl;
cout << "c1 = ";
print(c1);
cout << "c2 = ";
print(c2);
cout << "c3 = ";
print(c3);
cout << "c4 = ";
print(c4);

where_iter = c2.begin();
++where_iter; // start at second element
c2.splice(where_iter, c1);
cout << "After splicing c1 into c2:" << endl;
cout << "c1 = ";
print(c1);
cout << "c2 = ";
print(c2);

first_iter = c3.begin();
c2.splice(where_iter, c3, first_iter);
cout << "After splicing the first element of c3 into c2:" << endl;
cout << "c3 = ";
print(c3);
cout << "c2 = ";
print(c2);

first_iter = c4.begin();
last_iter = c4.end();
// set up to get the middle elements
++first_iter;
--last_iter;
c2.splice(where_iter, c4, first_iter, last_iter);
cout << "After splicing a range of c4 into c2:" << endl;
cout << "c4 = ";
print(c4);
cout << "c2 = ";
print(c2);
}

```

```

Beginning state of lists:c1 = 2 elements: (10) (11)c2 = 3 elements: (20) (21) (22)c3 = 2 elements: (30)
(31)c4 = 4 elements: (40) (41) (42) (43)After splicing c1 into c2:c1 = 0 elements:c2 = 5 elements: (20) (10)
(11) (21) (22)After splicing the first element of c3 into c2:c3 = 1 elements: (31)c2 = 6 elements: (20) (10)
(11) (30) (21) (22)After splicing a range of c4 into c2:c4 = 2 elements: (40) (43)c2 = 8 elements: (20) (10)
(11) (30) (41) (42) (21) (22)

```

list::swap

Exchanges the elements of two lists.

```

void swap(list<Type, Allocator>& right);
friend void swap(list<Type, Allocator>& left, list<Type, Allocator>& right)

```

Parameters

right

The list providing the elements to be swapped, or the list whose elements are to be exchanged with those of the list *left*.

left

A list whose elements are to be exchanged with those of the list *right*.

Example

```
// list_swap.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1, c2, c3;
    list<int>::iterator c1_Iter;

    c1.push_back( 1 );
    c1.push_back( 2 );
    c1.push_back( 3 );
    c2.push_back( 10 );
    c2.push_back( 20 );
    c3.push_back( 100 );

    cout << "The original list c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    c1.swap( c2 );

    cout << "After swapping with c2, list c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    swap( c1, c3 );

    cout << "After swapping with c3, list c1 is:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;
}
```

```
The original list c1 is: 1 2 3
After swapping with c2, list c1 is: 10 20
After swapping with c3, list c1 is: 100
```

list::unique

Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from a list.

```
void unique();

template <class BinaryPredicate>
void unique(BinaryPredicate pred);
```

Parameters

pred

The binary predicate used to compare successive elements.

Remarks

This function assumes that the list is sorted, so that all duplicate elements are adjacent. Duplicates that are not

adjacent will not be deleted.

The first member function removes every element that compares equal to its preceding element.

The second member function removes every element that satisfies the predicate function *pred* when compared with its preceding element. You can use any of the binary function objects declared in the <functional> header for the argument *pred* or you can create your own.

Example

```
// list_unique.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int> c1;
    list<int>::iterator c1_Iter, c2_Iter, c3_Iter;
    not_equal_to<int> mypred;

    c1.push_back( -10 );
    c1.push_back( 10 );
    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 20 );
    c1.push_back( -10 );

    cout << "The initial list is c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << " " << *c1_Iter;
    cout << endl;

    list<int> c2 = c1;
    c2.unique( );
    cout << "After removing successive duplicate elements, c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << " " << *c2_Iter;
    cout << endl;

    list<int> c3 = c2;
    c3.unique( mypred );
    cout << "After removing successive unequal elements, c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
        cout << " " << *c3_Iter;
    cout << endl;
}
```

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

list::value_type

A type that represents the data type stored in a list.

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter *Type*.

Example

```
// list_value_type.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
    using namespace std;
    list<int>::value_type AnInt;
    AnInt = 44;
    cout << AnInt << endl;
}
```

44

See also

[<list>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<locale>

11/9/2018 • 5 minutes to read • [Edit Online](#)

Defines template classes and functions that C++ programs can use to encapsulate and manipulate different cultural conventions regarding the representation and formatting of numeric, monetary, and calendric data, including internationalization support for character classification and string collation.

Syntax

```
#include <locale>
```

Functions

FUNCTION	DESCRIPTION
has_facet	Tests if a particular facet is stored in a specified locale.
isalnum	Tests whether an element in a locale is an alphabetic or a numeric character.
isalpha	Tests whether an element in a locale is alphabetic character.
isctrl	Tests whether an element in a locale is a control character.
isdigit	Tests whether an element in a locale is a numeric character.
isgraph	Tests whether an element in a locale is an alphanumeric or punctuation character.
islower	Tests whether an element in a locale is lower case.
isprint	Tests whether an element in a locale is a printable character.
ispunct	Tests whether an element in a locale is a punctuation character.
isspace	Tests whether an element in a locale is a whitespace character.
isupper	Tests whether an element in a locale is upper case.
isxdigit	Tests whether an element in a locale is a character used to represent a hexadecimal number.
tolower	Converts a character to lower case.
toupper	Converts a character to upper case.

FUNCTION	DESCRIPTION
use_facet	Returns a reference to a facet of a specified type stored in a locale.

Classes

CLASS	DESCRIPTION
codecvt	A template class that provides a facet used to convert between internal and external character encodings.
codecvt_base	A base class for the codecvt class that is used to define an enumeration type referred to as <code>result</code> , used as the return type for the facet member functions to indicate the result of a conversion.
codecvt_byname	A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning conversions.
collate	A collate template class that provides a facet that handles string sorting conventions.
collate_byname	A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning string sorting conventions.
ctype	A template class that provides a facet that is used to classify characters, convert from upper- and lowercase and between the native character set and that set used by the locale.
ctype<char>	A class that is an explicit specialization of template class <code>ctype<CharType></code> to type char , describing an object that can serve as a locale facet to characterize various properties of a character of type char .
ctype_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
ctype_byname	A derived template class that describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.
locale	A class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.
messages	A template class that describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.

CLASS	DESCRIPTION
messages_base	A base class that describes an int type for the catalog of messages.
messages_byname	A derived template class that describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.
money_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
money_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type CharType to monetary values.
money_put	A template class that describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type CharType .
moneypunct	A template class that describes an object that can serve as a locale facet to describe the sequences of type CharType used to represent a monetary input field or a monetary output field.
moneypunct_byname	A derived template class that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting monetary input or output fields.
num_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type CharType to numeric values.
num_put	A template class that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type CharType .
numpunct	A template class that describes an object that can serve as a local facet to describe the sequences of type CharType used to represent information about the formatting and punctuation of numeric and Boolean expressions.
numpunct_byname	A derived template class that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.
time_base	A class that serves as a base class for facets of template class time_get , defining just the enumerated type dateorder and several constants of this type.
time_get	A template class that describes an object that can serve as a locale facet to control conversions of sequences of type CharType to time values.

CLASS	DESCRIPTION
time_get_byname	A derived template class that describes an object that can serve as a locale facet of type <code>time_get<CharType, InputIterator></code> .
time_put	A template class that describes an object that can serve as a locale facet to control conversions of time values to sequences of type CharType .
time_put_byname	A derived template class that describes an object that can serve as a locale facet of type <code>time_put<CharType, OutputIterator></code> .
wbuffer_convert Class	Describes a stream buffer that controls the transmission of elements to and from a byte stream buffer.
wstring_convert Class	A template class that performs conversions between a wide string and a byte string.

See also

[Code Pages](#)

[Locale Names, Languages, and Country/Region Strings](#)

[Thread Safety in the C++ Standard Library](#)

<locale> functions

10/31/2018 • 15 minutes to read • [Edit Online](#)

has_facet	isalnum	isalpha
iscntrl	isdigit	isgraph
islower	isprint	ispunct
isspace	isupper	isxdigit
tolower	toupper	use_facet

has_facet

Tests if a particular facet is stored in a specified locale.

```
template <class Facet>
bool has_facet(const locale& Loc);
```

Parameters

Loc

The locale to be tested for the presence of a facet.

Return Value

true if the locale has the facet tested for; **false** if it does not.

Remarks

The template function is useful for checking whether nonmandatory facets are listed in a locale before `use_facet` is called to avoid the exception that would be thrown if it were not present.

Example

```
// locale_has_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result = has_facet <ctype<char> > ( loc );
    cout << result << endl;
}
```

isalnum

Tests whether an element in a locale is an alphabetic or a numeric character.

```
template <class CharType>
bool isalnum(CharType Ch, const locale& Loc)
```

Parameters

Ch

The alphanumeric element to be tested.

Loc

The locale containing the alphanumeric element to be tested.

Return Value

true if the element tested is alphanumeric; **false** if it is not.

Example

```
// locale_isalnum.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isalnum ( 'L', loc);
    bool result2 = isalnum ( '@', loc);
    bool result3 = isalnum ( '3', loc);

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "alphanumeric." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not alphanumeric." << endl;

    if ( result2 )
        cout << "The character '@' in the locale is "
              << "alphanumeric." << endl;
    else
        cout << "The character '@' in the locale is "
              << " not alphanumeric." << endl;

    if ( result3 )
        cout << "The character '3' in the locale is "
              << "alphanumeric." << endl;
    else
        cout << "The character '3' in the locale is "
              << " not alphanumeric." << endl;
}
```

```
The character 'L' in the locale is alphanumeric.
The character '@' in the locale is not alphanumeric.
The character '3' in the locale is alphanumeric.
```

isalpha

Tests whether an element in a locale is an alphabetic character.

```
template <class CharType>
bool isalpha(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the alphabetic element to be tested.

Return Value

true if the element tested is alphabetic; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc).is(ctype< CharType>:: alpha, Ch)`.

Example

```
// locale_isalpha.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isalpha ( 'L', loc);
    bool result2 = isalpha ( '@', loc);
    bool result3 = isalpha ( '3', loc);

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "alphabetic." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not alphabetic." << endl;

    if ( result2 )
        cout << "The character '@' in the locale is "
              << "alphabetic." << endl;
    else
        cout << "The character '@' in the locale is "
              << " not alphabetic." << endl;

    if ( result3 )
        cout << "The character '3' in the locale is "
              << "alphabetic." << endl;
    else
        cout << "The character '3' in the locale is "
              << " not alphabetic." << endl;
}
```

isctrl

Tests whether an element in a locale is a control character.


```
template <class CharType>
bool iscntrl(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a control character; **false** if it is not.

Remarks

The template function returns `use_facet<ctype< CharType>> (&Loc).is(ctype< CharType>::cntrl, Ch)`.

Example

```
// locale_iscntrl.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = iscntrl ( 'L', loc );
    bool result2 = iscntrl ( '\n', loc );
    bool result3 = iscntrl ( '\t', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a control character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a control character." << endl;

    if ( result2 )
        cout << "The character-set 'backslash-n' in the locale\n is "
              << "a control character." << endl;
    else
        cout << "The character-set 'backslash-n' in the locale\n is "
              << " not a control character." << endl;

    if ( result3 )
        cout << "The character-set 'backslash-t' in the locale\n is "
              << "a control character." << endl;
    else
        cout << "The character-set 'backslash-n' in the locale \n is "
              << " not a control character." << endl;
}
```

isdigit

Tests whether an element in a locale is a numeric character.

```
template <class CharType>
bool isdigit(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a numeric character; **false** if it is not.

Remarks

The template function returns `use_facet<ctype< CharType>> (&Loc).is(ctype< CharType>::digit, Ch)`.

Example

```
// locale_is_digit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isdigit ( 'L', loc );
    bool result2 = isdigit ( '@', loc );
    bool result3 = isdigit ( '3', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a numeric character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a numeric character." << endl;

    if ( result2 )
        cout << "The character '@' in the locale is "
              << "a numeric character." << endl;
    else
        cout << "The character '@' in the locale is "
              << " not a numeric character." << endl;

    if ( result3 )
        cout << "The character '3' in the locale is "
              << "a numeric character." << endl;
    else
        cout << "The character '3' in the locale is "
              << " not a numeric character." << endl;
}
```

isgraph

Tests whether an element in a locale is an alphanumeric or punctuation character.

```
template <class CharType>
bool isgraph(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is an alphanumeric or a punctuation character; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> > (Loc). is(ctype< CharType>:: graph, Ch)`.

Example

```
// locale_is_graph.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isgraph ( 'L', loc );
    bool result2 = isgraph ( '\\t', loc );
    bool result3 = isgraph ( '.', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is\\n "
              << "an alphanumeric or punctuation character." << endl;
    else
        cout << "The character 'L' in the locale is\\n "
              << " not an alphanumeric or punctuation character." << endl;

    if ( result2 )
        cout << "The character 'backslash-t' in the locale is\\n "
              << "an alphanumeric or punctuation character." << endl;
    else
        cout << "The character 'backslash-t' in the locale is\\n "
              << "not an alphanumeric or punctuation character." << endl;

    if ( result3 )
        cout << "The character '.' in the locale is\\n "
              << "an alphanumeric or punctuation character." << endl;
    else
        cout << "The character '.' in the locale is\\n "
              << " not an alphanumeric or punctuation character." << endl;
}
```

islower

Tests whether an element in a locale is lower case.

```
template <class CharType>
bool islower(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a lowercase character; **false** if it is not.

Remarks

The template function returns `use_facet<ctype< CharType>> (&Loc).is(ctype< CharType>::lower, Ch)`.

Example

```
// locale_islower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = islower ( 'L', loc );
    bool result2 = islower ( 'n', loc );
    bool result3 = islower ( '3', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a lowercase character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a lowercase character." << endl;

    if ( result2 )
        cout << "The character 'n' in the locale is "
              << "a lowercase character." << endl;
    else
        cout << "The character 'n' in the locale is "
              << " not a lowercase character." << endl;

    if ( result3 )
        cout << "The character '3' in the locale is "
              << "a lowercase character." << endl;
    else
        cout << "The character '3' in the locale is "
              << " not a lowercase character." << endl;
}
```

isprint

Tests whether an element in a locale is a printable character.

```
template <class CharType>
bool isprint(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a printable; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> > (Loc). is(ctype< CharType>:: print, Ch)`.

Example

```
// locale_isprint.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );

    bool result1 = isprint ( 'L', loc );
    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a printable character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a printable character." << endl;

    bool result2 = isprint( '\\t', loc );
    if ( result2 )
        cout << "The character 'backslash-t' in the locale is "
              << "a printable character." << endl;
    else
        cout << "The character 'backslash-t' in the locale is "
              << " not a printable character." << endl;

    bool result3 = isprint( '\\n', loc );
    if ( result3 )
        cout << "The character 'backslash-n' in the locale is "
              << "a printable character." << endl;
    else
        cout << "The character 'backslash-n' in the locale is "
              << " not a printable character." << endl;
}
```

ispunct

Tests whether an element in a locale is a punctuation character.

```
template <class CharType>
bool ispunct(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a punctuation character; **false** if it is not.

Remarks

The template function returns `use_facet<ctype< CharType>> (&Loc).is(ctype< CharType>::punct, Ch)`.

Example

```
// locale_ispunct.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = ispunct ( 'L', loc );
    bool result2 = ispunct ( ';', loc );
    bool result3 = ispunct ( '*', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a punctuation character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a punctuation character." << endl;

    if ( result2 )
        cout << "The character ';' in the locale is "
              << "a punctuation character." << endl;
    else
        cout << "The character ';' in the locale is "
              << " not a punctuation character." << endl;

    if ( result3 )
        cout << "The character '*' in the locale is "
              << "a punctuation character." << endl;
    else
        cout << "The character '*' in the locale is "
              << " not a punctuation character." << endl;
}
```

isspace

Tests whether an element in a locale is a whitespace character.

```
template <class CharType>
bool isspace(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a whitespace character; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc).is(ctype< CharType>::space, Ch)`.

Example

```
// locale_isspace.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isspace ( 'L', loc );
    bool result2 = isspace ( '\n', loc );
    bool result3 = isspace ( ' ', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a whitespace character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a whitespace character." << endl;

    if ( result2 )
        cout << "The character 'backslash-n' in the locale is "
              << "a whitespace character." << endl;
    else
        cout << "The character 'backslash-n' in the locale is "
              << " not a whitespace character." << endl;

    if ( result3 )
        cout << "The character ' ' in the locale is "
              << "a whitespace character." << endl;
    else
        cout << "The character ' ' in the locale is "
              << " not a whitespace character." << endl;
}
```

isupper

Tests whether an element in a locale is in upper case.

```
template <class CharType>
bool isupper(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is an uppercase character; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc).is(ctype< CharType>::upper, Ch)`.

Example

```
// locale_isupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isupper ( 'L', loc );
    bool result2 = isupper ( 'n', loc );
    bool result3 = isupper ( '3', loc );

    if ( result1 )
        cout << "The character 'L' in the locale is "
              << "a uppercase character." << endl;
    else
        cout << "The character 'L' in the locale is "
              << " not a uppercase character." << endl;

    if ( result2 )
        cout << "The character 'n' in the locale is "
              << "a uppercase character." << endl;
    else
        cout << "The character 'n' in the locale is "
              << " not a uppercase character." << endl;

    if ( result3 )
        cout << "The character '3' in the locale is "
              << "a uppercase character." << endl;
    else
        cout << "The character '3' in the locale is "
              << " not a uppercase character." << endl;
}
```

isxdigit

Tests whether an element in a locale is a character used to represent a hexadecimal number.

```
template <class CharType>
bool isxdigit(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

true if the element tested is a character used to represent a hexadecimal number; **false** if it is not.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc).is(ctype< CharType>::xdigit, Ch)`.

Hexadecimal digits use base 16 to represent numbers, using the numbers 0 through 9 plus case-insensitive letters A through F to represent the decimal numbers 0 through 15.

Example

```
// locale_isxdigit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = isxdigit ( '5', loc );
    bool result2 = isxdigit ( 'd', loc );
    bool result3 = isxdigit ( 'q', loc );

    if ( result1 )
        cout << "The character '5' in the locale is "
              << "a hexadecimal digit-character." << endl;
    else
        cout << "The character '5' in the locale is "
              << " not a hexadecimal digit-character." << endl;

    if ( result2 )
        cout << "The character 'd' in the locale is "
              << "a hexadecimal digit-character." << endl;
    else
        cout << "The character 'd' in the locale is "
              << " not a hexadecimal digit-character." << endl;

    if ( result3 )
        cout << "The character 'q' in the locale is "
              << "a hexadecimal digit-character." << endl;
    else
        cout << "The character 'q' in the locale is "
              << " not a hexadecimal digit-character." << endl;
}
```

tolower

Converts a character to lower case.

```
template <class CharType>
CharType tolower(CharType Ch, const locale& Loc)
```

Parameters

Ch

The character to be converted to lower case.

Loc

The locale containing the character to be converted.

Return Value

The character converted to lower case.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc). tolower(Ch)`.

Example

```
// locale_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    char result1 = tolower ( 'H', loc );
    cout << "The lower case of 'H' in the locale is: "
         << result1 << "." << endl;
    char result2 = tolower ( 'h', loc );
    cout << "The lower case of 'h' in the locale is: "
         << result2 << "." << endl;
    char result3 = tolower ( '$', loc );
    cout << "The lower case of '$' in the locale is: "
         << result3 << "." << endl;
}
```

toupper

Converts a character to upper case.

```
template <class CharType>
CharType toupper(CharType Ch, const locale& Loc)
```

Parameters

Ch

The character to be converted to upper case.

Loc

The locale containing the character to be converted.

Return Value

The character converted to upper case.

Remarks

The template function returns `use_facet< ctype< CharType> >(Loc). toupper(Ch)`.

Example

```
// locale_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    char result1 = toupper ( 'h', loc );
    cout << "The upper case of 'h' in the locale is: "
         << result1 << "." << endl;
    char result2 = toupper ( 'H', loc );
    cout << "The upper case of 'H' in the locale is: "
         << result2 << "." << endl;
    char result3 = toupper ( '$', loc );
    cout << "The upper case of '$' in the locale is: "
         << result3 << "." << endl;
}
```

use_facet

Returns a reference to a facet of a specified type stored in a locale.

```
template <class Facet>
const Facet& use_facet(const locale& Loc);
```

Parameters

Loc

The const locale containing the type of facet being referenced.

Return Value

A reference to the facet of class `Facet` contained within the argument locale.

Remarks

The reference to the facet returned by the template function remains valid as long as any copy of the containing locale exists. If no such facet object of class `Facet` is listed in the argument locale, the function throws a `bad_cast` exception.

Example

```

// locale_use_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );
    bool result1 = use_facet<ctype<char> > ( loc1 ).is(
        ctype_base::alpha, 'a'
    );
    bool result2 = use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha, '!'
    );

    if ( result1 )
        cout << "The character 'a' in locale loc1 is alphabetic."
            << endl;
    else
        cout << "The character 'a' in locale loc1 is not alphabetic."
            << endl;

    if ( result2 )
        cout << "The character '!' in locale loc2 is alphabetic."
            << endl;
    else
        cout << "The character '!' in locale loc2 is not alphabetic."
            << endl;
}

```

```

The character 'a' in locale loc1 is alphabetic.
The character '!' in locale loc2 is not alphabetic.

```

See also

[<locale>](#)

codecvt Class

10/31/2018 • 16 minutes to read • [Edit Online](#)

A template class that describes an object that can serve as a locale facet. It is able to control conversions between a sequence of values used to encode characters within the program and a sequence of values used to encode characters outside the program.

Syntax

```
template <class CharType, class Byte, class StateType>
class codecvt : public locale::facet, codecvt_base;
```

Parameters

CharType

The type used within a program to encode characters.

Byte

A type used to encode characters outside a program.

StateType

A type that can be used to represent intermediate states of a conversion between internal and external types of character representations.

Remarks

The template class describes an object that can serve as a [locale facet](#), to control conversions between a sequence of values of type *CharType* and a sequence of values of type *Byte*. The class *StateType* characterizes the transformation -- and an object of class *StateType* stores any necessary state information during a conversion.

The internal encoding uses a representation with a fixed number of bytes per character, usually either type **char** or type **wchar_t**.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The template versions of [do_in](#) and [do_out](#) always return `codecvt_base::noconv`.

The C++ Standard Library defines several explicit specializations:

```
template<>
codecvt<wchar_t, char, mbstate_t>
```

converts between **wchar_t** and **char** sequences.

```
template<>
codecvt<char16_t, char, mbstate_t>
```

converts between `char16_t` sequences encoded as UTF-16 and **char** sequences encoded as UTF-8.

```
template<>
codecv<char32_t, char, mbstate_t>
```

converts between `char32_t` sequences encoded as UTF-32 (UCS-4) and **char** sequences encoded as UTF-8.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>codecvt</code>	The constructor for objects of class <code>codecvt</code> that serves as a locale facet to handle conversions.

Typedefs

TYPE NAME	DESCRIPTION
<code>extern_type</code>	A character type that is used for external representations.
<code>intern_type</code>	A character type that is used for internal representations.
<code>state_type</code>	A character type that is used to represent intermediate states during conversions between internal and external representations.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>always_noconv</code>	Tests whether no conversions need be done.
<code>do_always_noconv</code>	A virtual function called to test whether no conversions need be done.
<code>do_encoding</code>	A virtual function that tests if the encoding of the <code>Byte</code> stream is state dependent, whether the ratio between the <code>Byte</code> s used and the <code>CharType</code> s produced is constant, and, if so, determines the value of that ratio.
<code>do_in</code>	A virtual function called to convert a sequence of internal <code>Byte</code> s to a sequence of external <code>CharType</code> s.
<code>do_length</code>	A virtual function that determines how many <code>Byte</code> s from a given sequence of external <code>Byte</code> s produce not more than a given number of internal <code>CharType</code> s and returns that number of <code>Byte</code> s.
<code>do_max_length</code>	A virtual function that returns the maximum number of external Bytes necessary to produce one internal <code>CharType</code> .
<code>do_out</code>	A virtual function called to convert a sequence of internal <code>CharType</code> s to a sequence of external Bytes.
<code>do_unshift</code>	A virtual function called to provide the <code>Byte</code> s needed in a state-dependent conversion to complete the last character in a sequence of <code>Byte</code> s.

MEMBER FUNCTION	DESCRIPTION
encoding	Tests if the encoding of the <code>Byte</code> stream is state dependent, whether the ratio between the <code>Byte</code> s used and the <code>CharType</code> s produced is constant, and, if so, determines the value of that ratio.
in	Converts an external representation of a sequence of <code>Byte</code> s to an internal representation of a sequence of <code>CharType</code> s.
length	Determines how many <code>Byte</code> s from a given sequence of external <code>Byte</code> s produce not more than a given number of internal <code>CharType</code> s and returns that number of <code>Byte</code> s.
max_length	Returns the maximum number of external <code>Byte</code> s necessary to produce one internal <code>CharType</code> .
out	Converts a sequence of internal <code>CharType</code> s to a sequence of external <code>Byte</code> s.
unshift	Provides the external <code>Byte</code> s needed in a state-dependent conversion to complete the last character in the sequence of <code>Byte</code> s.

Requirements

Header: <locale>

Namespace: std

codecvt::always_noconv

Tests whether no conversions need be done.

```
bool always_noconv() const throw();
```

Return Value

A Boolean value that is **true** if no conversions need be done; **false** is at least one needs to be done.

Remarks

The member function returns [do_always_noconv](#).

Example

```
// codecvt_always_noconv.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    bool result1 = use_facet<codecvt<char, char, mbstate_t> >
        ( loc ).always_noconv( );

    if ( result1 )
        cout << "No conversion is needed." << endl;
    else
        cout << "At least one conversion is required." << endl;

    bool result2 = use_facet<codecvt<wchar_t, char, mbstate_t> >
        ( loc ).always_noconv( );

    if ( result2 )
        cout << "No conversion is needed." << endl;
    else
        cout << "At least one conversion is required." << endl;
}
```

```
No conversion is needed.
At least one conversion is required.
```

codecvt::codecvt

The constructor for objects of class `codecvt` that serves as a locale facet to handle conversions.

```
explicit codecvt(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- 2: These values are not defined.

The constructor initializes its `locale::facet` base object with **locale::facet**(`_Refs`).

codecvt::do_always_noconv

A virtual function called to test whether no conversions need be done.

```
virtual bool do_always_noconv() const throw();
```

Return Value

The protected virtual member function returns **true** only if every call to `do_in` or `do_out` returns `noconv`.

The template version always returns **true**.

Example

See the example for `always_noconv`, which calls `do_always_noconv`.

codecvt::do_encoding

A virtual function that tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte`s used and the `CharType`s produced is constant and, if so, determines the value of that ratio.

```
virtual int do_encoding() const throw();
```

Return Value

The protected virtual member function returns:

- -1, if the encoding of sequences of type `extern_type` is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- N , if the encoding involves only sequences of length N

Example

See the example for `encoding`, which calls `do_encoding`.

codecvt::do_in

A virtual function called to convert a sequence of external `Byte`s to a sequence of internal `CharType`s.

```
virtual result do_in(  
    StateType& _State,  
    const Byte* first1,  
    const Byte* last1,  
    const Byte*& next1,  
    CharType* first2,  
    CharType* last2,  
    CharType*& next2,) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the sequence to be converted.

last1

Pointer to the end of the sequence to be converted.

next1

Pointer beyond the end of the converted sequence, to the first unconverted character.

first2

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Pointer to the `CharType` that comes after the last converted `CharType`, to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success, or failure of the operation. The function returns:

- `codecvt_base::error` if the source sequence is ill formed.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination is not large enough, for the conversion to succeed.

Remarks

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for [in](#), which calls `do_in`.

codecvt::do_length

A virtual function that determines how many `Byte`s from a given sequence of external `Byte`s produce not more than a given number of internal `CharType`s and returns that number of `Byte`s.

```
virtual int do_length(  
    const StateType& _State,  
    const Byte* first1,  
    const Byte* last1,  
    size_t _Len2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the external sequence.

last1

Pointer to the end of the external sequence.

_Len2

The maximum number of `Byte`s that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *_Len2*, defined by the external source sequence at [`first1`, `last1`).

Remarks

The protected virtual member function effectively calls `do_in` (`_State`, `first1`, `last1`, `next1`, `_Buf`, `_Buf` + `_Len2`, `next2`) for *_State* (a copy of state), some buffer `_Buf`, and pointers `next1` and `next2`.

It then returns `next2` - `buf`. Thus, it counts the maximum number of conversions, not greater than *_Len2*, defined

by the source sequence at [`first1` , `last1`).

The template version always returns the lesser of *last1 - first1* and *_Len2*.

Example

See the example for [length](#), which calls `do_length` .

codecvt::do_max_length

A virtual function that returns the maximum number of external `Byte` s necessary to produce one internal `CharType` .

```
virtual int do_max_length() const throw();
```

Return Value

The maximum number of `Byte` s necessary to produce one `CharType` .

Remarks

The protected virtual member function returns the largest permissible value that can be returned by [do_length](#)(`first1` , `last1` , 1) for arbitrary valid values of *first1* and *last1*.

Example

See the example for [max_length](#), which calls `do_max_length` .

codecvt::do_out

A virtual function called to convert a sequence of internal `CharType` s to a sequence of external `Byte` s.

```
virtual result do_out(  
    StateType& _State,  
    const CharType* first1,  
    const CharType* last1,  
    const CharType*& next1,  
    Byte* first2,  
    Byte* last2,  
    Byte*& next2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the sequence to be converted.

last1

Pointer to the end of the sequence to be converted.

next1

Reference to a pointer to the first unconverted `CharType` , after the last `CharType` converted.

first2

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Reference to a pointer to the first unconverted `Byte`, after the last `Byte` converted.

Return Value

The function returns:

- `codecvt_base::error` if the source sequence is ill formed.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination is not large enough for the conversion to succeed.

Remarks

`_State` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for `out`, which calls `do_out`.

codecvt::do_unshift

A virtual function called to provide the `Byte`s needed in a state-dependent conversion to complete the last character in a sequence of `Byte`s.

```
virtual result do_unshift(
    StateType& _State,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first2

Pointer to the first position in the destination range.

last2

Pointer to the last position in the destination range.

next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- `codecvt_base::error` if `_State` represents an invalid state
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the destination is not large enough for the conversion to succeed

Remarks

The protected virtual member function tries to convert the source element `CharType` (0) to a destination sequence that it stores within [`first2`, `last2`), except for the terminating element `Byte` (0). It always stores in `next2` a pointer to the first unaltered element in the destination sequence.

`_State` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Typically, converting the source element `CharType` (0) leaves the current state in the initial conversion state.

Example

See the example for [unshift](#), which calls `do_unshift`.

codecvt::encoding

Tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte`s used and the `CharType`s produced is constant, and, if so, determines the value of that ratio.

```
int encoding() const throw();
```

Return Value

If the return value is positive then that value is the constant number of `Byte` characters required to produce the `CharType` character.

The protected virtual member function returns:

- -1, if the encoding of sequences of type `extern_type` is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- *N*, if the encoding involves only sequences of length *N*.

Remarks

The member function returns [do_encoding](#).

Example

```
// codecvt_encoding.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    int result1 = use_facet<codecvt<char, char, mbstate_t> > ( loc ).encoding ( );
    cout << result1 << endl;
    result1 = use_facet<codecvt<wchar_t, char, mbstate_t> > ( loc ).encoding( );
    cout << result1 << endl;
    result1 = use_facet<codecvt<char, wchar_t, mbstate_t> > ( loc ).encoding( );
    cout << result1 << endl;
}
```

```
1
1
1
```

codecvt::extern_type

A character type that is used for external representations.

```
typedef Byte extern_type;
```

Remarks

The type is a synonym for the template parameter `Byte`.

codecvt::in

Converts an external representation of a sequence of `Byte`s to an internal representation of a sequence of `CharType` S.

```
result in(  
    StateType& _State,  
    const Byte* first1,  
    const Byte* last1,  
    const Byte*& next1,  
    CharType* first2,  
    CharType* last2,  
    CharType*& next2,) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the sequence to be converted.

last1

Pointer to the end of the sequence to be converted.

next1

Pointer beyond the end of the converted sequence to the first unconverted character.

first2

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Pointer to the `CharType` that comes after the last converted `CharType` to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success or failure of the operation. The function returns:

- `codecvt_base::error` if the source sequence is ill formed.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination is not large enough for the conversion to succeed.

Remarks

`_State` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. After a partial conversion, `_State` must be set so as to allow the conversion to resume when new characters arrive.

The member function returns `do_in(_State, _First1, last1, next1, First2, _Llast2, next2)`.

Example

```
// codecvt_in.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
    char* pszExt = "This is the string to be converted!";
    wchar_t pwszInt [LEN+1];
    memset(&pwszInt[0], 0, (sizeof(wchar_t))*(LEN+1));
    const char* pszNext;
    wchar_t* pwszNext;
    mbstate_t state = {0};
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
        ( loc ).in( state,
            pszExt, &pszExt[strlen(pszExt)], pszNext,
            pwszInt, &pwszInt[strlen(pszExt)], pwszNext );
    pwszInt[strlen(pszExt)] = 0;
    wcout << ( (res!=codecvt_base::error) ? L"It worked! " : L"It didn't work! " )
    << L"The converted string is:\n ["
    << &pwszInt[0]
    << L"]" << endl;
    exit(-1);
}
```

```
It worked! The converted string is:
[This is the string to be converted!]
```

codecvt::intern_type

A character type that is used for internal representations.

```
typedef CharType intern_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

codecvt::length

Determines how many `Byte`s from a given sequence of external `Byte`s produce not more than a given number of internal `CharType`s and returns that number of `Byte`s.

```
int length(
    const StateType& _State,
    const Byte* first1,
    const Byte* last1,
    size_t _Len2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the external sequence.

last1

Pointer to the end of the external sequence.

_Len2

The maximum number of Bytes that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *_Len2*, defined by the external source sequence at [*first1* , *last1*).

Remarks

The member function returns `do_length(_State, first1, last1, _Len2)`.

Example

```
// codecvt_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
    char* pszExt = "This is the string whose length is to be measured!";
    mbstate_t state = {0};
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
        ( loc ).length( state,
            pszExt, &pszExt[strlen(pszExt)], LEN );
    cout << "The length of the string is: ";
    wcout << res;
    cout << "." << endl;
    exit(-1);
}
```

The length of the string is: 50.

codecvt::max_length

Returns the maximum number of external `Byte`s necessary to produce one internal `CharType`.

```
int max_length() const throw();
```


Return Value

The maximum number of `Byte`s necessary to produce one `CharType`.

Remarks

The member function returns `do_max_length`.

Example

```
// codecvt_max_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc( "C");//English_Britain" );//German_Germany
    int res = use_facet<codecvt<char, char, mbstate_t> >
        ( loc ).max_length( );
    wcout << res << endl;
}
```

1

codecvt::out

Converts a sequence of internal `CharType`s to a sequence of external `Byte`s.

```
result out(
    StateType& _State,
    const CharType* first1,
    const CharType* last1,
    const CharType*& next1,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the sequence to be converted.

last1

Pointer to the end of the sequence to be converted.

next1

Reference to a pointer to the first unconverted `CharType` after the last `CharType` converted.

first2

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Reference to a pointer to the first unconverted `Byte` after the last converted `Byte` .

Return Value

The member function returns `do_out(_State , first1 , last1 , next1 , first2 , last2 , next2)`.

Remarks

For more information, see [codecvt::do_out](#).

Example

```
// codecvt_out.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
#include <wchar.h>
using namespace std;
#define LEN 90
int main( )
{
    char pszExt[LEN+1];
    wchar_t *pwszInt = L"This is the wchar_t string to be converted.";
    memset( &pszExt[0], 0, ( sizeof( char ) )*( LEN+1 ) );
    char* pszNext;
    const wchar_t* pwszNext;
    mbstate_t state;
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet<codecvt<wchar_t, char, mbstate_t> >
        ( loc ).out( state,
        pwszInt, &pwszInt[wcslen( pwszInt )], pwszNext ,
        pszExt, &pszExt[wcslen( pwszInt )], pszNext );
    pszExt[wcslen( pwszInt )] = 0;
    cout << ( ( res!=codecvt_base::error ) "It worked: " : "It didn't work: " )
    << "The converted string is:\n ["
    << &pszExt[0]
    << "]" << endl;
}
```

```
It worked: The converted string is:
[This is the wchar_t string to be converted.]
```

codecvt::state_type

A character type that is used to represent intermediate states during conversions between internal and external representations.

```
typedef StateType state_type;
```

Remarks

The type is a synonym for the template parameter `StateType` .

codecvt::unshift

Provides the `Byte` s needed in a state-dependent conversion to complete the last character in a sequence of `Byte` s.

```
result unshift(
    StateType& _State,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

_State

The conversion state that is maintained between calls to the member function.

first2

Pointer to the first position in the destination range.

last2

Pointer to the last position in the destination range.

next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- `codecvt_base::error` if state represents an invalid state.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the destination is not large enough for the conversion to succeed.

Remarks

The protected virtual member function tries to convert the source element `CharType` (0) to a destination sequence that it stores within [`first2`, `last2`), except for the terminating element `Byte` (0). It always stores in *next2* a pointer to the first unaltered element in the destination sequence.

_State must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Typically, converting the source element `CharType` (0) leaves the current state in the initial conversion state.

The member function returns `do_unshift(_State, first2, last2, next2)`.

See also

[<locale>](#)

[Code Pages](#)

[Locale Names, Languages, and Country/Region Strings](#)

[Thread Safety in the C++ Standard Library](#)

codecvt_base Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A base class for the `codecvt` class that is used to define an enumeration type referred to as `result`, used as the return type for the facet member functions to indicate the result of a conversion.

Syntax

```
class codecvt_base : public locale::facet {
public:
    enum result {ok, partial, error, noconv};
    codecvt_base( size_t _Refs = 0);
    bool always_noconv() const;
    int max_length() const;
    int encoding() const;
    ~codecvt_base()

protected:
    virtual bool do_always_noconv() const;
    virtual int do_max_length() const;
    virtual int do_encoding() const;
};
```

Remarks

The class describes an enumeration common to all specializations of template class `codecvt`. The enumeration `result` describes the possible return values from `do_in` or `do_out`:

- `ok` if the conversion between internal and external character encodings succeeds.
- `partial` if the destination is not large enough for the conversion to succeed.
- `error` if the source sequence is ill formed.
- `noconv` if the function performs no conversion.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

codecvt_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning conversions.

Syntax

```
template <class CharType, class Byte, class StateType>
class codecvt_byname: public codecvt<CharType, Byte, StateType> {
public:
    explicit codecvt_byname(
        const char* _Locname,
        size_t _Refs = 0);
```

```
    explicit codecvt_byname(
        const string& _Locname,
        size_t _Refs = 0);
```

```
protected:
    virtual ~codecvt_byname();

};
```

Parameters

_Locname

A named locale.

_Refs

An initial reference count.

Remarks

Byname facets are automatically created when a named locale is constructed.

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with `codecvt<CharType, Byte, StateType>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

collate Class

10/31/2018 • 7 minutes to read • [Edit Online](#)

A template class that describes an object that can serve as a locale facet to control the ordering and grouping of characters within a string, comparisons between them and the hashing of strings.

Syntax

```
template <class CharType>
class collate : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`. In some languages, characters are grouped and treated as a single character, and in others, individual characters are treated as if they were two characters. The collating services provided by the collate class provide the way to sort these cases.

Constructors

CONSTRUCTOR	DESCRIPTION
collate	The constructor for objects of class <code>collate</code> that serves as a locale facet to handle string sorting conventions.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that describes a character of type <code>CharType</code> .
string_type	A type that describes a string of type <code>basic_string</code> containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
compare	Compares two character sequences according to their facet-specific rules for equality or inequality.
do_compare	A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.
do_hash	A virtual function called to determine the hash value of sequences according to their facet-specific rules.

MEMBER FUNCTION	DESCRIPTION
do_transform	A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.
hash	Determines the hash value of sequence according to their facet-specific rules.
transform	Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

Requirements

Header: <locale>

Namespace: std

collate::char_type

A type that describes a character of type `CharType`.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

collate::collate

The constructor for objects of class collate that serves as a locale facet to handle string sorting conventions.

```
public:
    explicit collate(
        size_t _Refs = 0);

protected:
    collate(
        const char* _Locname,
        size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

_Locname

The name of the locale.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.

- > 1: These values are not defined.

The constructor initializes its base object with **locale::facet**(`_Refs`).

collate::compare

Compares two character sequences according to their facet-specific rules for equality or inequality.

```
int compare(const CharType* first1,
            const CharType* last1,
            const CharType* first2,
            const CharType* last2) const;
```

Parameters

first1

Pointer to the first element in the first sequence to be compared.

last1

Pointer to the last element in the first sequence to be compared.

first2

Pointer to the first element in the second sequence to be compared.

last2

Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or, if no unequal pairs exist, but the first sequence is shorter.

The member function returns `do_compare`(`first1`, `last1`, `first2`, `last2`).

Example


```

// collate_compare.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
    locale loc ( "German_germany" );
    _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s, it comes before z in the German
    alphabet
    _TCHAR * s2 = _T("Das ist weizzz.");
    int result1 = use_facet<collate<_TCHAR> > ( loc ).
        compare ( s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << result1 << endl;

    locale loc2 ( "C" );
    int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
        compare ( s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << result2 << endl;
}

```

collate::do_compare

A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.

```

virtual int do_compare(const CharType* first1,
    const CharType* last1,
    const CharType* first2,
    const CharType* last2) const;

```

Parameters

first1

Pointer to the first element in the first sequence to be compared.

last1

Pointer to the last element in the first sequence to be compared.

first2

Pointer to the first element in the second sequence to be compared.

last2

Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The protected virtual member function compares the sequence at [*first1*, *Last1*)* with the sequence at [*first2*, *last2*). It compares values by applying `operator<` between pairs of corresponding elements of type `CharType`. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences or if no unequal pairs exist but the first sequence is shorter.

Example

See the example for [collate::compare](#), which calls `do_compare`.

collate::do_hash

A virtual function called to determine the hash value of sequences according to their facet-specific rules.

```
virtual long do_hash(const CharType* first, const CharType* last) const;
```

Parameters

first

A pointer to the first character in the sequence whose has value is to be determined.

last

A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type **long** for the sequence.

Remarks

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

See the example for [hash](#), which calls `do_hash`.

collate::do_transform

A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
virtual string_type do_transform(const CharType* first, const CharType* last) const;
```

Parameters

first

A pointer to the first character in the sequence to be converted.

last

A pointer to the last character in the sequence to be converted.

Return Value

A string that is the transformed character sequence.

Remarks

The protected virtual member function returns an object of class [string_type](#) whose controlled sequence is a copy of the sequence [`first`, `last`). If a class derived from `collate< CharType >` overrides [do_compare](#), it should also override `do_transform` to match. When passed to `collate::compare`, two transformed strings should yield the same result that you would get from passing the untransformed strings to `compare` in the derived class.

Example

See the example for [transform](#), which calls `do_transform`.

collate::hash

Determines the hash value of sequence according to their facet-specific rules.

```
long hash(const CharType* first, const CharType* last) const;
```

Parameters

first

A pointer to the first character in the sequence whose has value is to be determined.

last

A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type **long** for the sequence.

Remarks

The member function returns `do_hash(first, last)`.

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

```
// collate_hash.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
    locale loc ( "German_germany" );
    _TCHAR * s1 = _T("\x00dfzz abc."); // \x00df is the German sharp-s (looks like beta), it comes before z in
the alphabet
    _TCHAR * s2 = _T("zzz abc."); // \x00df is the German sharp-s (looks like beta), it comes before z in the
alphabet

    long r1 = use_facet< collate<_TCHAR> > ( loc ).
        hash (s1, &s1[_tcslen( s1 )-1 ] );
    long r2 = use_facet< collate<_TCHAR> > ( loc ).
        hash (s2, &s2[_tcslen( s2 )-1 ] );
    cout << r1 << " " << r2 << endl;
}
```

```
541187293 551279837
```

collate::string_type

A type that describes a string of type `basic_string` containing characters of type `CharType`.

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of template class `basic_string` whose objects can store copies of the source sequence.

Example

For an example of how to declare and use `string_type`, see [transform](#).

collate::transform

Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
string_type transform(const CharType* first, const CharType* last) const;
```

Parameters

first

A pointer to the first character in the sequence to be converted.

last

A pointer to the last character in the sequence to be converted.

Return Value

A string that contains the transformed character sequence.

Remarks

The member function returns [do_transform](#)(`first`, `last`).

Example

```
// collate_transform.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
    locale loc ( "German_Germany" );
    _TCHAR* s1 = _T("\x00dfzz abc.");
    // \x00df is the German sharp-s (looks like beta),
    // it comes before z in the alphabet
    _TCHAR* s2 = _T("zzz abc.");

    collate<_TCHAR>::string_type r1; // OK for typedef
    r1 = use_facet< collate<_TCHAR> > ( loc ).
        transform (s1, &s1[_tcslen( s1 )-1 ]);

    cout << r1 << endl;

    basic_string<_TCHAR> r2 = use_facet< collate<_TCHAR> > ( loc ).
        transform (s2, &s2[_tcslen( s2 )-1 ]);

    cout << r2 << endl;

    int result1 = use_facet<collate<_TCHAR> > ( loc ).compare
        (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );

    cout << _tcscmp(r1.c_str( ),r2.c_str( )) << result1
        << _tcscmp(s1,s2) <<endl;
}
```

See also

[<locale>](#)

[Thread Safety in the C++ Standard Library](#)

collate_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A derived template class that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning string sorting conventions.

Syntax

```
template <class CharType>
class collate_byname : public collate<CharType> {
public:
    explicit collate_byname(
        const char* _Locname,
        size_t _Refs = 0);

    explicit collate_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual ~collate_byname();

};
```

Parameters

_Locname

A named locale.

_Refs

An initial reference count.

Remarks

The template class describes an object that can serve as a [locale facet](#) of type `collate<CharType>`. Its behavior is determined by the [named](#) locale *_Locname*. Each constructor initializes its base object with `collate<CharType>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

ctype Class

10/31/2018 • 17 minutes to read • [Edit Online](#)

A class that provides a facet that is used to classify characters, convert from upper and lower cases, and convert between the native character set and that set used by the locale.

Syntax

```
template <class CharType>
class ctype : public ctype_base;
```

Parameters

CharType

The type used within a program to encode characters.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`. Classification criteria are provided a nested bitmask type in the base class `ctype_base`.

The C++ Standard Library defines two explicit specializations of this template class:

- `ctype<char>`, an explicit specialization whose differences are described separately. For more information, see [ctype<char> Class](#).
- `ctype<wchar_t>`, which treats elements as wide characters.

Other specializations of template class `ctype<CharType>`:

- Convert a value *ch* of type *CharType* to a value of type **char** with the expression `(char)ch`.
- Convert a value *byte* of type **char** to a value of type *CharType* with the expression `CharType(byte)`.

All other operations are performed on **char** values in the same way as for the explicit specialization

`ctype<char>`.

Constructors

CONSTRUCTOR	DESCRIPTION
ctype	Constructor for objects of class <code>ctype</code> that serve as locale facets for characters.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that describes a character used by a locale.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>do_is</code>	A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.
<code>do_narrow</code>	A virtual function called to convert a character of type <code>CharType</code> used by a locale to the corresponding character of type char in the native character set.
<code>do_scan_is</code>	A virtual function called to locate the first character in a range that matches a specified mask.
<code>do_scan_not</code>	A virtual function called to locate the first character in a range that does not match a specified mask.
<code>do_tolower</code>	A virtual function called to convert a character or a range of characters to their lower case.
<code>do_toupper</code>	A virtual function called to convert a character or a range of characters to upper case.
<code>do_widen</code>	A virtual function called to converts a character of type char in the native character set to the corresponding character of type <code>CharType</code> used by a locale.
<code>is</code>	Tests whether a single character has a particular attribute, or classifies the attributes of each character in a range and stores them in an array.
<code>narrow</code>	Converts a character of type <code>CharType</code> used by a locale to the corresponding character of type <code>char</code> in the native character set.
<code>scan_is</code>	Locates the first character in a range that matches a specified mask.
<code>scan_not</code>	Locates the first character in a range that does not match a specified mask.
<code>tolower</code>	Converts a character or a range of characters to lower case.
<code>toupper</code>	Converts a character or a range of characters to upper case.
<code>widen</code>	Converts a character of type char in the native character set to the corresponding character of type <code>CharType</code> used by a locale.

Requirements

Header: `<locale>`

Namespace: `std`

`ctype::char_type`

A type that describes a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

Example

See the member function [widen](#) for an example that uses `char_type` as a return value.

ctype::ctype

Constructor for objects of class `ctype` that serve as locale facets for characters.

```
explicit ctype(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its `locale::facet` base object with **`locale::facet`**(`_Refs`).

ctype::do_is

A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.

```
virtual bool do_is(
    mask maskVal,
    CharType ch) const;

virtual const CharType *do_is(
    const CharType* first,
    const CharType* last,
    mask* dest) const;
```

Parameters

maskVal

The mask value for which the character is to be tested.

ch

The character whose attributes are to be tested.

first

A pointer to the first character in the range whose attributes are to be classified.

last

A pointer to the character immediately following the last character in the range whose attributes are to be classified.

dest

A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns a Boolean value that is **true** if the character tested has the attribute described by the mask value; **false** if it fails to have the attribute.

The second member function returns an array containing the mask values characterizing the attributes of each of the characters in the range.

Remarks

The mask values classifying the attributes of the characters are provided by the class `ctype_base`, from which `ctype` derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the logical bitwise operators (`|`, `&`, `^`, `~`).

Example

See the example for `is`, which calls `do_is`.

`ctype::do_narrow`

A virtual function called to convert a character of type `CharType` used by a locale to the corresponding character of type **char** in the native character set.

```
virtual char do_narrow(
    CharType ch,
    char default = '\0') const;

virtual const CharType* do_narrow(
    const CharType* first,
    const CharType* last,
    char default,
    char* dest) const;
```

Parameters

ch

The character of type `CharType` used by the locale to be converted.

default

The default value to be assigned by the member function to characters of type `CharType` that do not have counterpart characters of type **char**.

first

A pointer to the first character in the range of characters to be converted.

last

A pointer to the character immediately following the last character in the range of characters to be converted.

dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the native character of type `char` that corresponds to the parameter character of type `CharType` or *default* if no counterpart is defined.

The second protected member function returns a pointer to the destination range of native characters converted from characters of type `CharType`.

Remarks

The second protected member template function stores in `dest [I]` the value `do_narrow (first [I], default)`, for `I` in the interval `[0, last - first)`.

Example

See the example for [narrow](#), which calls `do_narrow`.

ctype::do_scan_is

A virtual function called to locate the first character in a range that matches a specified mask.

```
virtual const CharType *do_scan_is(  
    mask maskVal,  
    const CharType* first,  
    const CharType* last) const;
```

Parameters

maskVal

The mask value to be matched by a character.

first

A pointer to the first character in the range to be scanned.

last

A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *last*.

Remarks

The protected member function returns the smallest pointer `ptr` in the range `[first , last)` for which `do_is(maskVal , * ptr)` is true.

Example

See the example for [scan_is](#), which calls `do_scan_is`.

ctype::do_scan_not

A virtual function called to locate the first character in a range that does not match a specified mask.

```
virtual const CharType *do_scan_not(  
    mask maskVal,  
    const CharType* first,  
    const CharType* last) const;
```

Parameters

maskVal

The mask value not to be matched by a character.

first

A pointer to the first character in the range to be scanned.

last

A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that doesn't match a specified mask. If no such value exists, the function returns *last*.

Remarks

The protected member function returns the smallest pointer `ptr` in the range [`first`, `last`) for which `do_is(maskVal, * ptr)` is false.

Example

See the example for [scan_not](#), which calls `do_scan_not`.

ctype::do_tolower

A virtual function called to convert a character or a range of characters to lower case.

```
virtual CharType do_tolower(CharType ch) const;

virtual const CharType *do_tolower(
    CharType* first,
    const CharType* last) const;
```

Parameters

ch

The character to be converted to lower case.

first

A pointer to the first character in the range of characters whose cases are to be converted.

last

A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the lowercase form of the parameter *ch*. If no lowercase form exists, it returns *ch*. The second protected member function returns *last*.

Remarks

The second protected member template function replaces each element `first` [`I`], for `I` in the interval [0, `last` - `first`), with `do_tolower` (`first` [`I`]).

Example

See the example for [tolower](#), which calls `do_tolower`.

ctype::do_toupper

A virtual function called to convert a character or a range of characters to upper case.

```
virtual CharType do_toupper(CharType ch) const;

virtual const CharType *do_toupper(
    CharType* first,
    const CharType* last) const;
```

Parameters

ch

The character to be converted to upper case.

first

A pointer to the first character in the range of characters whose cases are to be converted.

last

A pointer to character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the uppercase form of the parameter *ch*. If no uppercase form exists, it returns *ch*. The second protected member function returns *last*.

Remarks

The second protected member template function replaces each element `first [I]`, for `I` in the interval `[0, last - first)`, with `do_toupper (first [I])`.

Example

See the example for [toupper](#), which calls `do_toupper`.

ctype::do_widen

A virtual function called to converts a character of type **char** in the native character set to the corresponding character of type `CharType` used by a locale.

```
virtual CharType do_widen(char byte) const;

virtual const char *do_widen(
    const char* first,
    const char* last,
    CharType* dest) const;
```

Parameters

byte

The character of type **char** in the native character set to be converted.

first

A pointer to the first character in the range of characters to be converted.

last

A pointer to the character immediately following the last character in the range of characters to be converted.

dest

A pointer to the first character of type `CharType` in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the character of type `CharType` that corresponds to the parameter character of native type **char**.

The second protected member function returns a pointer to the destination range of characters of type `CharType` used by a locale converted from native characters of type **char**.

Remarks

The second protected member template function stores in `dest [I]` the value `do_widen (first [I])`, for `I` in the interval `[0, last - first)`.

Example

See the example for [widen](#), which calls `do_widen`.

ctype::is

Tests whether a single character has a particular attribute or classifies the attributes of each character in a range and stores them in an array.

```
bool is(mask maskVal, CharType ch) const;

const CharType *is(
    const CharType* first,
    const CharType* last,
    mask* dest) const;
```

Parameters

maskVal

The mask value for which the character is to be tested.

ch

The character whose attributes are to be tested.

first

A pointer to the first character in the range whose attributes are to be classified.

last

A pointer to the character immediately following the last character in the range whose attributes are to be classified.

dest

A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns **true** if the character tested has the attribute described by the mask value; **false** if it fails to have the attribute.

The second member function returns a pointer to the last character in the range whose attributes are to be classified.

Remarks

The mask values classifying the attributes of the characters are provided by the class [ctype_base Class](#), from which `ctype` derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the logical bitwise operators (`|`, `&`, `^`, `~`).

Example

```

// ctype_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main() {
    locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );

    if (use_facet<ctype<char> > ( loc1 ).is( ctype_base::alpha, 'a' ))
        cout << "The character 'a' in locale loc1 is alphabetic."
              << endl;
    else
        cout << "The character 'a' in locale loc1 is not alphabetic."
              << endl;

    if (use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha, '!' ))
        cout << "The character '!' in locale loc2 is alphabetic."
              << endl;
    else
        cout << "The character '!' in locale loc2 is not alphabetic."
              << endl;

    char *string = "Hello, my name is John!";
    ctype<char>::mask maskarray[30];
    use_facet<ctype<char> > ( loc2 ).is(
        string, string + strlen(string), maskarray );
    for (unsigned int i = 0; i < strlen(string); i++) {
        cout << string[i] << ": "
              << (maskarray[i] & ctype_base::alpha ? "alpha"
                  : "not alpha")
              << endl;
    }
}

```

ctype::narrow

Converts characters of type `CharType` used by a locale to the corresponding characters of type **char** in the native character set.

```

char narrow(CharType ch, char default = '\0') const;

const CharType* narrow(
    const CharType* first,
    const CharType* last,
    char default,
    char* dest) const;

```

Parameters

ch

The character of type `CharType` used by the locale to be converted.

default

The default value to be assigned by the member function to characters of type `CharType` that do not have counterpart characters of type **char**.

first

A pointer to the first character in the range of characters to be converted.

last

A pointer to the character immediately following the last character in the range of characters to be converted.

dest

A const pointer to the first character of type **char** in the destination range that stores the converted range of characters.

Return Value

The first member function returns the native character of type **char** that corresponds to the parameter character of type `CharType default` if not counterpart is defined.

The second member function returns a pointer to the destination range of native characters converted from characters of type `CharType`.

Remarks

The first member function returns `do_narrow(ch, default)`. The second member function returns `do_narrow (first, last, default, dest)`. Only the basic source characters are guaranteed to have a unique inverse image `CharType` under `narrow`. For these basic source characters, the following invariant holds: `narrow (widen (c), 0) == c`.

Example

```
// ctype_narrow.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "english" );
    wchar_t *str1 = L"\x0392fhello everyone";
    char str2 [16];
    bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).narrow
        ( str1, str1 + wcslen(str1), 'X', &str2[0] ) != 0); // C4996
    str2[wcslen(str1)] = '\0';
    wcout << str1 << endl;
    cout << &str2[0] << endl;
}
```

Xhello everyone

ctype::scan_is

Locates the first character in a range that matches a specified mask.

```
const CharType *scan_is(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal

The mask value to be matched by a character.

first

A pointer to the first character in the range to be scanned.

last

A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *last*.

Remarks

The member function returns `do_scan_is(maskVal, first, last)`.

Example

```
// ctype_scan_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "German_Germany" );

    char *string = "Hello, my name is John!";

    const char* i = use_facet<ctype<char> > ( loc1 ).scan_is
        ( ctype_base::punct, string, string + strlen(string) );
    cout << "The first punctuation is \"" << *i << "\" at position: "
        << i - string << endl;
}
```

The first punctuation is "," at position: 5

ctype::scan_not

Locates the first character in a range that does not match a specified mask.

```
const CharType *scan_not(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal

The mask value not to be matched by a character.

first

A pointer to the first character in the range to be scanned.

last

A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does not match a specified mask. If no such value exists, the function returns *last*.

Remarks

The member function returns `do_scan_not(maskVal, first, last)`.

Example

```
// ctype_scan_not.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "German_Germany" );

    char *string = "Hello, my name is John!";

    const char* i = use_facet<ctype<char> > ( loc1 ).scan_not
        ( ctype_base::alpha, string, string + strlen(string) );
    cout << "First nonalpha character is \"" << *i << "\" at position: "
        << i - string << endl;
}
```

```
First nonalpha character is ", " at position: 5
```

ctype::tolower

Converts a character or a range of characters to lower case.

```
CharType tolower(CharType ch) const;

const CharType *tolower(CharType* first, const CharType* last) const;
```

Parameters

ch

The character to be converted to lower case.

first

A pointer to the first character in the range of characters whose cases are to be converted.

last

A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the lowercase form of the parameter *ch*. If no lowercase form exists, it returns *ch*.

The second member function returns *last*.

Remarks

The first member function returns `do_tolower(ch)`. The second member function returns `do_tolower(first, last)`.

Example

```
// ctype_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "German_Germany" );

    char string[] = "HELLO, MY NAME IS JOHN";

    use_facet<ctype<char> > ( loc1 ).tolower
        ( string, string + strlen(string) );
    cout << "The lowercase string is: " << string << endl;
}
```

The lowercase string is: hello, my name is john

ctype::toupper

Converts a character or a range of characters to upper case.

```
CharType toupper(CharType ch) const;
const CharType *toupper(CharType* first, const CharType* last) const;
```

Parameters

ch

The character to be converted to uppercase.

first

A pointer to the first character in the range of characters whose cases are to be converted.

last

A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the uppercase form of the parameter *ch*. If no uppercase form exists, it returns *ch*.

The second member function returns *last*.

Remarks

The first member function returns `do_toupper(ch)`. The second member function returns `do_toupper(first , last)`.

Example

```
// ctype_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "German_Germany" );

    char string[] = "Hello, my name is John";

    use_facet<ctype<char> > ( loc1 ).toupper
        ( string, string + strlen(string) );
    cout << "The uppercase string is: " << string << endl;
}
```

The uppercase string is: HELLO, MY NAME IS JOHN

ctype::widen

Converts a character of type **char** in the native character set to the corresponding character of type `CharType` used by a locale.

```
CharType widen(char byte) const;
const char *widen(const char* first, const char* last, CharType* dest) const;
```

Parameters

byte

The character of type `char` in the native character set to be converted.

first

A pointer to the first character in the range of characters to be converted.

last

A pointer to the character immediately following the last character in the range of characters to be converted.

dest

A pointer to the first character of type `CharType` in the destination range that stores the converted range of characters.

Return Value

The first member function returns the character of type `CharType` that corresponds to the parameter character of native type **char**.

The second member function returns a pointer to the destination range of characters of type `CharType` used by a locale converted from native characters of type **char**.

Remarks

The first member function returns `do_widen(byte)`. The second member function returns `do_widen(first, last, dest)`.

Example

```
// ctype_widen.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
    locale loc1 ( "English" );
    char *str1 = "Hello everyone!";
    wchar_t str2 [16];
    bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).widen
        ( str1, str1 + strlen(str1), &str2[0] ) != 0); // C4996
    str2[strlen(str1)] = '\0';
    cout << str1 << endl;
    wcout << &str2[0] << endl;

    ctype<wchar_t>::char_type charT;
    charT = use_facet<ctype<char> > ( loc1 ).widen( 'a' );
}
```

```
Hello everyone!
Hello everyone!
```

See also

[<locale>](#)

[Thread Safety in the C++ Standard Library](#)

ctype<char> Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class is an explicit specialization of template class `ctype<CharType>` to type **char**, describing an object that can serve as a locale facet to characterize various properties of a character of type **char**.

Syntax

```
template <>
class ctype<char>
: public ctype_base
{
public:
    typedef char _Elem;
    typedef _Elem char_type;
    bool is(
        mask _Maskval,
        _Elem _Ch) const;

    const _Elem* is(
        const _Elem* first,
        const _Elem* last,
        mask* dest) const;

    const _Elem* scan_is(
        mask _Maskval,
        const _Elem* first,
        const _Elem* last) const;

    const _Elem* scan_not(
        mask _Maskval,
        const _Elem* first,
        const _Elem* last) const;

    _Elem tolower(
        _Elem _Ch) const;

    const _Elem* tolower(
        _Elem* first,
        const _Elem* last) const;

    _Elem toupper(
        _Elem _Ch) const;

    const _Elem* toupper(
        _Elem* first,
        const _Elem* last) const;

    _Elem widen(
        char _Byte) const;

    const _Elem* widen(
        const char* first,
        const char* last,
        _Elem* dest) const;

    const _Elem* _Widen_s(
        const char* first,
        const char* last,
        _Elem* dest,
        size_t dest_size) const;
```

```

_Elem narrow(
    _Elem _Ch,
    char _Dflt = '\0') const;

const _Elem* narrow(
    const _Elem* first,
    const _Elem* last,
    char _Dflt,
    char* dest) const;

const _Elem* _Narrow_s(
    const _Elem* first,
    const _Elem* last,
    char _Dflt,
    char* dest,
    size_t dest_size) const;

static locale::id& id;
explicit ctype(
    const mask* _Table = 0,
    bool _Deletetable = false,
    size_t _Refs = 0);

protected:
    virtual ~ctype();
    //other protected members
};

```

Remarks

The explicit specialization differs from the template class in several ways:

- An object of class `ctype< char >` stores a pointer to the first element of a ctype mask table, an array of `UCHAR_MAX + 1` elements of type `ctype_base::mask`. It also stores a Boolean object that indicates whether the array should be deleted (using `operator delete[]`) when the `ctype< Elem >` object is destroyed.
- Its sole public constructor lets you specify `tab`, the ctype mask table, and `del`, the Boolean object that is true if the array should be deleted when the `ctype< char >` object is destroyed, as well as the reference-count parameter `refs`.
- The protected member function `table` returns the stored ctype mask table.
- The static member object `table_size` specifies the minimum number of elements in a ctype mask table.
- The protected static member function `classic_table` (returns the ctype mask table appropriate to the "C" locale.
- There are no protected virtual member functions `do_is`, `do_scan_is`, or `do_scan_not`. The corresponding public member functions perform the equivalent operations themselves.

The member functions `do_narrow` and `do_widen` copy elements unaltered.

Requirements

Header: `<locale>`

Namespace: `std`

See also

[facet Class](#)

[ctype_base Class](#)

[Thread Safety in the C++ Standard Library](#)

ctype_base Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as a base class for facets of template class [ctype](#). A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.

Syntax

```
struct ctype_base : public locale::facet
{
    enum
    {
        alnum,
        alpha,
        cntrl,
        digit,
        graph,
        lower,
        print,
        punct,
        space,
        upper,
        xdigit
    };
    typedef short mask;

    ctype_base( size_t _Refs = 0 );
    ~ctype_base();
};
```

Remarks

It defines an enumeration mask. Each enumeration constant characterizes a different way to classify characters, as defined by the functions with similar names declared in the header `<ctype.h>`. The constants are:

- **space** (function [isspace](#))
- **print** (function [isprint](#))
- **cntrl** (function [iscntrl](#))
- **upper** (function [isupper](#))
- **lower** (function [islower](#))
- **digit** (function [isdigit](#))
- **punct** (function [ispunct](#))
- **xdigit** (function [isxdigit](#))
- **alpha** (function [isalpha](#))
- **alnum** (function [isalnum](#))
- **graph** (function [isgraph](#))

You can characterize a combination of classifications by ORing these constants. In particular, it is always true that **alnum** == (**alpha** | **digit**) and **graph** == (**alnum** | **punct**).

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

ctype_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The derived template class describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.

Syntax

```
template <class _Elem>
class ctype_byname : public ctype<_Elem>
{
public:
    explicit ctype_byname(
        const char* _Locname,
        size_t _Refs = 0);

    explicit ctype_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual __CLR_OR_THIS_CALL ~ctype_byname();

};
```

Remarks

Its behavior is determined by the named locale `_Locname`. Each constructor initializes its base object with `ctype<CharType>(_Refs)` or the equivalent for base class `ctype<char>`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

locale Class

3/20/2019 • 13 minutes to read • [Edit Online](#)

The class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.

Syntax

```
class locale;
```

Remarks

A facet is a pointer to an object of a class derived from class [facet](#) that has a public object of the form:

```
static locale::id id;
```

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the [locale categories](#) traditionally managed in the Standard C Library by the function `setlocale`.

Category collate (LC_COLLATE) includes the facets:

```
collate<char>  
collate<wchar_t>
```

Category ctype (LC_CTYPE) includes the facets:

```
ctype<char>  
ctype<wchar_t>  
codecvt<char, char, mbstate_t>  
codecvt<wchar_t, char, mbstate_t>  
codecvt<char16_t, char, mbstate_t>  
codecvt<char32_t, char, mbstate_t>
```

Category monetary (LC_MONETARY) includes the facets:

```
moneypunct<char, false>  
moneypunct<wchar_t, false>  
moneypunct<char, true>  
moneypunct<wchar_t, true>  
money_get<char, istreambuf_iterator<char>>  
money_get<wchar_t, istreambuf_iterator<wchar_t>>  
money_put<char, ostreambuf_iterator<char>>  
money_put<wchar_t, ostreambuf_iterator<wchar_t>>
```

Category numeric (LC_NUMERIC) includes the facets:

```
num_get<char, istreambuf_iterator<char>>
num_get<wchar_t, istreambuf_iterator<wchar_t>>
num_put<char, ostreambuf_iterator<char>>
num_put<wchar_t, ostreambuf_iterator<wchar_t>>
numpunct<char>
numpunct<wchar_t>
```

Category time (LC_TIME) includes the facets:

```
time_get<char, istreambuf_iterator<char>>
time_get<wchar_t, istreambuf_iterator<wchar_t>>
time_put<char, ostreambuf_iterator<char>>
time_put<wchar_t, ostreambuf_iterator<wchar_t>>
```

Category messages (LC_MESSAGES) includes the facets:

```
messages<char>
messages<wchar_t>
```

(The last category is required by Posix, but not the C Standard.)

Some of these predefined facets are used by the iostreams classes, to control the conversion of numeric values to and from text sequences.

An object of class `locale` also stores a locale name as an object of class `string`. Using an invalid locale name to construct a locale facet or a locale object throws an object of class `runtime_error`. The stored locale name is `"*"` if the locale object cannot be certain that a C-style locale corresponds exactly to that represented by the object. Otherwise, you can establish a matching locale within the Standard C Library, for the locale object `Loc`, by calling `setlocale(LC_ALL, Loc.name().c_str())`.

In this implementation, you can also call the static member function:

```
static locale empty();
```

to construct a locale object that has no facets. It is also a transparent locale; if the template functions `has_facet` and `use_facet` cannot find the requested facet in a transparent locale, they consult first the global locale and then, if that is transparent, the classic locale. Thus, you can write:

```
cout.imbue(locale::empty());
```

Subsequent insertions to `cout` are mediated by the current state of the global locale. You can even write:

```
locale loc(locale::empty(),
           locale::classic(),
           locale::numeric);

cout.imbue(loc);
```

Numeric formatting rules for subsequent insertions to `cout` remain the same as in the C locale, even as the global locale supplies changing rules for inserting dates and monetary amounts.

Constructors

CONSTRUCTOR	DESCRIPTION
locale	Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale.

Typedefs

TYPE NAME	DESCRIPTION
category	An integer type that provides bitmask values to denote standard facet families.

Member functions

MEMBER FUNCTION	DESCRIPTION
combine	Inserts a facet from a specified locale into a target locale.
name	Returns the stored locale name.

Static Functions

classic	The static member function returns a locale object that represents the classic C locale.
global	Resets the default local for the program.

Operators

OPERATOR	DESCRIPTION
operator!=	Tests two locales for inequality.
operator()	Compares two <code>basic_string</code> objects.
operator==	Tests two locales for equality.

Classes

CLASS	DESCRIPTION
facet	A class that serves as the base class for all locale facets.
id	The member class provides a unique facet identification used as an index for looking up facets in a locale.

Requirements

Header: <locale>

Namespace: std

locale::category

An integer type that provides bitmask values to denote standard facet families.

```
typedef int category;
static const int collate = LC_COLLATE;
static const int ctype = LC_CTYPE;
static const int monetary = LC_MONETARY;
static const int numeric = LC_NUMERIC;
static const int time = LC_TIME;
static const int messages = LC_MESSAGES;
static const int all = LC_ALL;
static const int none = 0;
```

Remarks

The type is a synonym for an **int** type that can represent a group of distinct elements of a bitmask type local to class locale or can be used to represent any of the corresponding C locale categories. The elements are:

- `collate`, corresponding to the C category LC_COLLATE
- `ctype`, corresponding to the C category LC_CTYPE
- `monetary`, corresponding to the C category LC_MONETARY
- `numeric`, corresponding to the C category LC_NUMERIC
- `time`, corresponding to the C category LC_TIME
- `messages`, corresponding to the Posix category LC_MESSAGES

In addition, two useful values are:

- `none`, corresponding to none of the C categories
- `all`, corresponding to the C union of all categories LC_ALL

You can represent an arbitrary group of categories by using `OR` with these constants, as in `monetary | time`.

locale::classic

The static member function returns a locale object that represents the classic C locale.

```
static const locale& classic();
```

Return Value

A reference to the C locale.

Remarks

The classic C locale is the U.S. English ASCII locale within the Standard C Library that is implicitly used in programs that are not internationalized.

Example

```
// locale_classic.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
    locale loc1( "german" );
    locale loc2 = locale::global( loc1 );
    cout << "The name of the previous locale is: " << loc2.name( )
        << "." << endl;
    cout << "The name of the current locale is: " << loc1.name( )
        << "." << endl;

    if (loc2 == locale::classic( ) )
        cout << "The previous locale was classic." << endl;
    else
        cout << "The previous locale was not classic." << endl;

    if (loc1 == locale::classic( ) )
        cout << "The current locale is classic." << endl;
    else
        cout << "The current locale is not classic." << endl;
}
```

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
The previous locale was classic.
The current locale is not classic.
```

locale::combine

Inserts a facet from a specified locale into a target locale.

```
template <class Facet>
locale combine(const locale& Loc) const;
```

Parameters

Loc

The locale containing the facet to be inserted into the target locale.

Return Value

The member function returns a locale object that replaces in or adds to ***this** the facet `Facet` listed in *Loc*.

Example


```

// locale_combine.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
    locale loc ( "German_germany" );
    _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s; it comes before z in the German
    alphabet
    _TCHAR * s2 = _T("Das ist weizzz.");
    int result1 = use_facet<collate<_TCHAR> > ( loc ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

    locale loc2 ( "C" );
    int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc2 ) << result2 << endl;

    locale loc3 = loc2.combine<collate<_TCHAR> > (loc);
    int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;
}

```

facet Class

A class that serves as the base class for all locale facets.

```

class facet {
protected:
    explicit facet(size_t _Refs = 0);
    virtual ~facet();
private:
    facet(const facet&)
    // not defined void operator=(const facet&)
    // not defined
};

```

Remarks

Note that you cannot copy or assign an object of class facet. You can construct and destroy objects derived from class `locale::facet` but not objects of the base class proper. Typically, you construct an object `_Myfac` derived from facet when you construct a locale, as in **localeloc**(`locale::classic` (), **new** `_Myfac`);

In such cases, the constructor for the base class facet should have a zero `_Refs` argument. When the object is no longer needed, it is deleted. Thus, you supply a nonzero `_Refs` argument only in those rare cases where you take responsibility for the lifetime of the object.

locale::global

Resets the default locale for the program. This affects the global locale for both C and C++.

```

static locale global(const locale& Loc);

```

Parameters

Loc

The locale to be used as the default locale by the program.

Return Value

The previous locale before the default locale was reset.

Remarks

At program startup, the global locale is the same as the classic locale. The `global()` function calls `setlocale(LC_ALL, loc.name. c_str())` to establish a matching locale in the Standard C library.

Example

```
// locale_global.cpp
// compile by using: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
    locale loc ( "German_germany" );
    locale loc1;
    cout << "The initial locale is: " << loc1.name( ) << endl;
    locale loc2 = locale::global ( loc );
    locale loc3;
    cout << "The current locale is: " << loc3.name( ) << endl;
    cout << "The previous locale was: " << loc2.name( ) << endl;
}
```

```
The initial locale is: C
The current locale is: German_Germany.1252
The previous locale was: C
```

id Class

The member class provides a unique facet identification used as an index for looking up facets in a locale.

```
class id
{
    protected:    id();
    private:      id(const id&)
    void operator=(const id&) // not defined
};
```

Remarks

The member class describes the static member object required by each unique locale facet. Note that you cannot copy or assign an object of class `id`.

locale::locale

Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale.

```

locale();

explicit locale(const char* Locname, category Cat = all);
explicit locale(const string& Locname);
locale( const locale& Loc);
locale(const locale& Loc, const locale& Other, category Cat);
locale(const locale& Loc, const char* Locname, category Cat);

template <class Facet>
locale(const locale& Loc, const Facet* Fac);

```

Parameters

Locname

Name of a locale.

Loc

A locale that is to be copied in constructing the new locale.

Other

A locale from which to select a category.

Cat

The category to be substituted into the constructed locale.

Fac

The facet to be substituted into the constructed locale.

Remarks

The first constructor initializes the object to match the global locale. The second and third constructors initialize all the locale categories to have behavior consistent with the locale name *Locname*. The remaining constructors copy *Loc*, with the exceptions noted:

```
locale(const locale& Loc, const locale& Other, category Cat);
```

replaces from *Other* those facets corresponding to a category *C* for which *C* & *Cat* is nonzero.

```
locale(const locale& Loc, const char* Locname, category Cat);
```

```
locale(const locale& Loc, const string& Locname, category Cat);
```

replaces from `locale(Locname, _All)` those facets corresponding to a category *C* for which *C* & *Cat* is nonzero.

```
template<class Facet> locale(const locale& Loc, Facet* Fac);
```

replaces in (or adds to) *Loc* the facet *Fac*, if *Fac* is not a null pointer.

If a locale name *Locname* is a null pointer or otherwise invalid, the function throws [runtime_error](#).

Example

```

// locale_locale.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( ) {

    // Second constructor
    locale loc ( "German_germany" );
    _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-s, it comes before z in the German
    alphabet
    _TCHAR * s2 = _T("Das ist weizzz.");
    int result1 = use_facet<collate<_TCHAR> > ( loc ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

    // The first (default) constructor
    locale loc2;
    int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc2 ) << result2 << endl;

    // Third constructor
    locale loc3 (loc2,loc, _M_COLLATE );
    int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;

    // Fourth constructor
    locale loc4 (loc2, "German_Germany", _M_COLLATE );
    int result4 = use_facet<collate<_TCHAR> > ( loc4 ).
        compare (s1, &s1[_tcslen( s1 )-1 ], s2, &s2[_tcslen( s2 )-1 ] );
    cout << isalpha (_T ( '\x00df' ), loc4 ) << result4 << endl;
}

```

locale::name

Returns the stored locale name.

```
string name() const;
```

Return Value

A string giving the name of the locale.

Example

```
// locale_name.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
    locale loc1( "german" );
    locale loc2 = locale::global( loc1 );
    cout << "The name of the previous locale is: "
         << loc2.name( ) << "." << endl;
    cout << "The name of the current locale is: "
         << loc1.name( ) << "." << endl;
}
```

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
```

locale::operator!=

Tests two locales for inequality.

```
bool operator!=(const locale& right) const;
```

Parameters

right

One of the locales to be tested for inequality.

Return Value

A Boolean value that is **true** if the locales are not copies of the same locale; **false** if the locales are copies of the same locale.

Remarks

Two locales are equal if they are the same locale, if one is a copy of the other, or if they have identical names.

Example

```
// locale_op_ne.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
    locale loc1( "German_Germany" );
    locale loc2( "German_Germany" );
    locale loc3( "English" );

    if ( loc1 != loc2 )
        cout << "locales loc1 (" << loc1.name( )
        << ") and\n loc2 (" << loc2.name( ) << ") are not equal." << endl;
    else
        cout << "locales loc1 (" << loc1.name( )
        << ") and\n loc2 (" << loc2.name( ) << ") are equal." << endl;

    if ( loc1 != loc3 )
        cout << "locales loc1 (" << loc1.name( )
        << ") and\n loc3 (" << loc3.name( ) << ") are not equal." << endl;
    else
        cout << "locales loc1 (" << loc1.name( )
        << ") and\n loc3 (" << loc3.name( ) << ") are equal." << endl;
}
```

```
locales loc1 (German_Germany.1252) and
loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252) and
loc3 (English_United States.1252) are not equal.
```

locale::operator()

Compares two `basic_string` objects.

```
template <class CharType, class Traits, class Allocator>
bool operator()(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right) const;
```

Parameters

left

The left string.

right

The right string.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The member function effectively executes:

```
const collate<CharType>& fac = use_fac<collate<CharType>>(*this);

return (fac.compare(left.begin(), left.end(), right.begin(), right.end()) < 0);
```

Thus, you can use a locale object as a function object.

Example

```
// locale_op_compare.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

int main( )
{
    using namespace std;
    wchar_t *sa = L"ztesting";
    wchar_t *sb = L"\0x00DFtesting";
    basic_string<wchar_t> a( sa );
    basic_string<wchar_t> b( sb );

    locale loc( "German_Germany" );
    cout << loc( a,b ) << endl;

    const collate<wchar_t>& fac = use_facet<collate<wchar_t>>( loc );
    cout << ( fac.compare( sa, sa + a.length( ),
        sb, sb + b.length( ) ) < 0 ) << endl;
}
```

```
0
0
```

locale::operator==

Tests two locales for equality.

```
bool operator==(const locale& right) const;
```

Parameters

right

One of the locales to be tested for equality.

Return Value

A Boolean value that is **true** if the locales are copies of the same locale; **false** if the locales are not copies of the same locale.

Remarks

Two locales are equal if they are the same locale, if one is a copy of the other, or if they have identical names.

Example

```

// locale_op_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
    locale loc1( "German_Germany" );
    locale loc2( "German_Germany" );
    locale loc3( "English" );

    if ( loc1 == loc2 )
        cout << "locales loc1 ( " << loc1.name( )
        << " )\n and loc2 ( " << loc2.name( ) << " ) are equal."
        << endl;
    else
        cout << "locales loc1 ( " << loc1.name( )
        << " )\n and loc2 ( " << loc2.name( ) << " ) are not equal."
        << endl;

    if ( loc1 == loc3 )
        cout << "locales loc1 ( " << loc1.name( )
        << " )\n and loc3 ( " << loc3.name( ) << " ) are equal."
        << endl;
    else
        cout << "locales loc1 ( " << loc1.name( )
        << " )\n and loc3 ( " << loc3.name( ) << " ) are not equal."
        << endl;
}

```

```

locales loc1 (German_Germany.1252)
and loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252)
and loc3 (English_United States.1252) are not equal.

```

See also

[<locale>](#)

[Code Pages](#)

[Locale Names, Languages, and Country/Region Strings](#)

[Thread Safety in the C++ Standard Library](#)

messages Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.

Currently, while the messages class is implemented, there are no messages.

Syntax

```
template <class CharType>
class messages : public messages_base;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

This facet basically opens a catalog of messages defined in the base class `messages_base`, retrieves the information required, and closes the catalog.

Constructors

CONSTRUCTOR	DESCRIPTION
messages	The message facet constructor function.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A character type that is used display messages.
string_type	A type that describes a string of type <code>basic_string</code> containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
close	Closes the message catalog.
do_close	A virtual function called to lose the message catalog.
do_get	A virtual function called to retrieve the message catalog.

MEMBER FUNCTION	DESCRIPTION
do_open	A virtual function called to open the message catalog.
get	Retrieves the message catalog.
open	Opens the message catalog.

Requirements

Header: <locale>

Namespace: std

messages::char_type

A character type that is used display messages.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

messages::close

Closes the message catalog.

```
void close(catalog _Catval) const;
```

Parameters

_Catval

The catalog to be closed.

Remarks

The member function calls [do_close](#)(*_Catval*).

messages::do_close

A virtual function called to lose the message catalog.

```
virtual void do_close(catalog _Catval) const;
```

Parameters

_Catval

The catalog to be closed.

Remarks

The protected member function closes the message catalog *_Catval*, which must have been opened by an earlier call to [do_open](#).

_Catval must be obtained from a previously opened catalog that is not closed.

Example

See the example for [close](#), which calls `do_close` .

messages::do_get

A virtual function called to retrieve the message catalog.

```
virtual string_type do_get(  
    catalog _Catval,  
    int _Set,  
    int _Message,  
    const string_type& _Dfault) const;
```

Parameters

_Catval

The identification value specifying the message catalog to be searched.

_Set

The first identified used to locate a message in a message catalog.

_Message

The second identified used to locate a message in a message catalog.

_Dfault

The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The protected member function tries to obtain a message sequence from the message catalog *_Catval*. It may make use of *_Set*, *_Message*, and *_Dfault* in doing so.

Example

See the example for [get](#), which calls `do_get` .

messages::do_open

A virtual function called to open the message catalog.

```
virtual catalog do_open(  
    const string& _Catname,  
    const locale& _Loc) const;
```

Parameters

_Catname

The name of the catalog to be searched.

_Loc

The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to [get](#).

Remarks

The protected member function tries to open a message catalog whose name is *_Catname*. It may make use of the *locale_Loc* in doing so

The return value should be used as the argument on a later call to [close](#).

Example

See the example for [open](#), which calls `do_open`.

messages::get

Retrieves the message catalog.

```
string_type get(
    catalog _CatVal,
    int _Set,
    int _Message,
    const string_type& _Dfault) const;
```

Parameters

_Catval

The identification value specifying the message catalog to be searched.

_Set

The first identified used to locate a message in a message catalog.

_Message

The second identified used to locate a message in a message catalog.

_Dfault

The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The member function returns [do_get](#)(`_Catval` , `_Set` , `_Message` , `_Dfault`).

messages::messages

The message facet constructor function.

```
explicit messages(
    size_t _Refs = 0);

protected: messages(
    const char* _Locname,
    size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

_Locname

The name of the locale.

Remarks

The possible values for the `_Refs` parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with `locale::facet(_Refs)`.

messages::open

Opens the message catalog.

```
catalog open(  
    const string& _Catname,  
    const locale& _Loc) const;
```

Parameters

`_Catname`

The name of the catalog to be searched.

`_Loc`

The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to [get](#).

Remarks

The member function returns `do_open(_Catname , _Loc)`.

messages::string_type

A type that describes a string of type `basic_string` containing characters of type `CharType`.

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of template class [basic_string](#) whose objects can store copies of the message sequences.

See also

[<locale>](#)

[messages_base](#) Class

[Thread Safety in the C++ Standard Library](#)

messages_base Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The base class describes an **int** type for the catalog of messages.

Syntax

```
struct messages_base : locale::facet {  
    typedef int catalog;  
    explicit messages_base(size_t _Refs = 0)  
};
```

Remarks

The type catalog is a synonym for type **int** that describes the possible return values from `messages::do_open`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

messages_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The derived template class describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.

Syntax

```
template <class CharType>
class messages_byname : public messages<CharType> {
public:
    explicit messages_byname(
        const char *_Locname,
        size_t _Refs = 0);

    explicit messages_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual ~messages_byname();

};
```

Parameters

_Locname

A named locale.

_Refs

An initial reference count.

Remarks

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with `messages<CharType>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

money_base Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class describes an enumeration and a structure common to all specializations of template class [moneypunct](#).

Syntax

```
struct pattern
{
    char field[_PATTERN_FIELD_SIZE];
};
```

Remarks

The enumeration `part` describes the possible values in elements of the array field in the structure pattern. The values of `part` are:

- `none` to match zero or more spaces or generate nothing.
- `sign` to match or generate a positive or negative sign.
- `space` to match zero or more spaces or generate a space.
- `symbol` to match or generate a currency symbol.
- `value` to match or generate a monetary value.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

money_get Class

10/31/2018 • 7 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to monetary values.

Syntax

```
template <class CharType, class InputIterator = istreambuf_iterator<CharType>>
class money_get : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

InputIterator

The type of iterator from which the get functions read their input.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **`id`**.

Constructors

CONSTRUCTOR	DESCRIPTION
money_get	The constructor for objects of type <code>money_get</code> that are used to extract numerical values from sequences representing monetary values.

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an input iterator.
string_type	A type that describes a string containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
do_get	A virtual function called to extracts a numerical value from a character sequence that represents a monetary value.

MEMBER FUNCTION	DESCRIPTION
get	Extracts a numerical value from a character sequence that represents a monetary value.

Requirements

Header: <locale>

Namespace: std

money_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

money_get::do_get

Virtual function called to extracts a numerical value from a character sequence that represents a monetary value.

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
    ios_base::iostate& State,
    long double& val) const virtual iter_type do_get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
    ios_base::iostate& State,
    string_type& val) const
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded or not.

val

A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

The first virtual protected member function tries to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty monetary input field. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (-), to represent the amount and stores the result in the `string_type` object *val*. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in *val* and sets `ios_base::failbit` in *State*. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in *State*.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally signed digit sequence to a value of type **long double** and stores that value in *val*.

The format of a monetary input field is determined by the `locale facet` *fac* returned by the effective call `use_facet < money_punct < CharType, intl > > (iosbase.getloc())`.

Specifically:

- **fac.neg_format** determines the order in which components of the field occur.
- **fac.curr_symbol** determines the sequence of elements that constitutes a currency symbol.
- **fac.positive_sign** determines the sequence of elements that constitutes a positive sign.
- **fac.negative_sign** determines the sequence of elements that constitutes a negative sign.
- **fac.grouping** determines how digits are grouped to the left of any decimal point.
- **fac.thousands_sep** determines the element that separates groups of digits to the left of any decimal point.
- **fac.decimal_point** determines the element that separates the integer digits from the fraction digits.
- **fac.frac_digits** determines the number of significant fraction digits to the right of any decimal point. When parsing a monetary amount with more fraction digits than are called for by `frac_digits`, `do_get` stops parsing after consuming at most `frac_digits` characters.

If the sign string (**fac.negative_sign** or **fac.positive_sign**) has more than one element, only the first element is matched where the element equal to **money_base::sign** appears in the format pattern (**fac.neg_format**). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If **iosbase.flags** & **showbase** is nonzero, the string **fac.curr_symbol** must match where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, if **money_base::symbol** occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is not matched. Otherwise, the currency symbol is optionally matched.

If no instances of **fac.thousands_sep** occur in the value portion of the monetary input field (where the element equal to **money_base::value** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by **fac.grouping** is enforced. Note that the resulting digit sequence represents an integer whose low-order **fac.frac_digits** decimal digits are considered to the right of the decimal point.

Arbitrary white space is matched where the element equal to **money_base::space** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element *ch* is considered white space if `use_facet < ctype < CharType > > (iosbase.getloc()). is(ctype_base::space, ch)` is **true**.

Example

See the example for [get](#), which calls `do_get`.

money_get::get

Extracts a numerical value from a character sequence that represents a monetary value.

```
iter_type get(iter_type first,
              iter_type last,
              bool Intl,
              ios_base& Iosbase,
              ios_base::iostate& State,
              long double& val) const;

iter_type get(iter_type first,
              iter_type last,
              bool Intl,
              ios_base& Iosbase,
              ios_base::iostate& State,
              string_type& val) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

State

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

val

A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

Both member functions return `do_get`(`first`, `last`, `Intl`, `Iosbase`, `State`, `val`).

Example

```

// money_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;

int main( )
{
    locale loc( "german_germany" );

    basic_stringstream< char > psz;
    psz << use_facet<moneypunct<char, 1> >(loc).curr_symbol() << "-1.000,56";
    basic_stringstream< char > psz2;
    psz2 << "-100056" << use_facet<moneypunct<char, 1> >(loc).curr_symbol();

    ios_base::iostate st = 0;
    long double fVal;

    psz.flags( psz.flags( )|ios_base::showbase );
    // Which forced the READING the currency symbol
    psz.imbue(loc);
    use_facet < money_get < char > >( loc ).
        get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
            basic_istream<char>::_Iter( 0 ), true, psz, st, fVal );

    if ( st & ios_base::failbit )
        cout << "money_get(" << psz.str( ) << ", intl = 1) FAILED"
            << endl;
    else
        cout << "money_get(" << psz.str( ) << ", intl = 1) = "
            << fVal/100.0 << endl;

    use_facet < money_get < char > >( loc ).
        get(basic_istream<char>::_Iter(psz2.rdbuf( )),
            basic_istream<char>::_Iter(0), false, psz2, st, fVal);

    if ( st & ios_base::failbit )
        cout << "money_get(" << psz2.str( ) << ", intl = 0) FAILED"
            << endl;
    else
        cout << "money_get(" << psz2.str( ) << ", intl = 0) = "
            << fVal/100.0 << endl;
};

```

money_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

money_get::money_get

The constructor for objects of type `money_get` that are used to extract numerical values from sequences representing monetary values.

```
explicit money_get(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::facet**(*_Refs*).

money_get::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of template class [basic_string](#).

See also

[<locale>](#)

[facet](#) Class

[Thread Safety in the C++ Standard Library](#)

money_put Class

11/2/2018 • 6 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type `CharType`.

Syntax

```
template <class CharType,  
         class OutputIterator = ostreambuf_iterator<CharType>>  
class money_put : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

OutputIterator

The type of iterator to which the monetary put functions write their output.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **`id`**.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>money_put</code>	The constructor for objects of type <code>money_put</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>iter_type</code>	A type that describes an output iterator.
<code>string_type</code>	A type that describes a string containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>do_put</code>	A virtual function called to convert either number or a string to a character sequence that represents a monetary value.
<code>put</code>	Converts either number or a string to a character sequence that represents a monetary value.

Requirements

Header: <locale>

Namespace: std

money_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

money_put::do_put

A virtual function called to convert either number or a string to a character sequence that represents a monetary value.

```
virtual iter_type do_put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    const string_type& val) const;

virtual iter_type do_put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

Parameters

next

An iterator addressing the first element of the inserted string.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill

A character which is used for spacing.

val

A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *next* to produce a monetary output field from the [string_type](#) object *val*. The sequence controlled by *val* must begin with one or

more decimal digits, optionally preceded by a minus sign (-), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts *val* to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the [locale facet](#) `fac` returned by the (effective) call `use_facet < money_punct < CharType, intl > > (iosbase.getloc)`.

Specifically:

- **fac.** [pos_format](#) determines the order in which components of the field are generated for a nonnegative value.
- **fac.** [neg_format](#) determines the order in which components of the field are generated for a negative value.
- **fac.** [curr_symbol](#) determines the sequence of elements to generate for a currency symbol.
- **fac.** [positive_sign](#) determines the sequence of elements to generate for a positive sign.
- **fac.** [negative_sign](#) determines the sequence of elements to generate for a negative sign.
- **fac.** [grouping](#) determines how digits are grouped to the left of any decimal point.
- **fac.** [thousands_sep](#) determines the element that separates groups of digits to the left of any decimal point.
- **fac.** [decimal_point](#) determines the element that separates the integer digits from any fraction digits.
- **fac.** [frac_digits](#) determines the number of significant fraction digits to the right of any decimal point.

If the sign string (**fac.** [negative_sign](#) or **fac.** [positive_sign](#)) has more than one element, only the first element is generated where the element equal to **money_base::sign** appears in the format pattern (**fac.** [neg_format](#) or **fac.** [pos_format](#)). Any remaining elements are generated at the end of the monetary output field.

If **iosbase.flags** & **showbase** is nonzero, the string **fac.** [curr_symbol](#) is generated where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by **fac.grouping** (its first element has the value `CHAR_MAX`), then no instances of **fac.** [thousands_sep](#) are generated in the value portion of the monetary output field (where the element equal to **money_base::value** appears in the format pattern). If **fac.** [frac_digits](#) is zero, then no instance of **fac.** [decimal_point](#) is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order **fac.** [frac_digits](#) decimal digits to the right of the decimal point.

Padding occurs as for any numeric output field, except that if **iosbase.flags** & **iosbase.internal** is nonzero, any internal padding is generated where the element equal to **money_base::space** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is **fill**.

The function calls **iosbase.width**(0) to reset the field width to zero.

Example

See the example for [put](#), where the virtual member function is called by **put**.

money_put::iter_type

A type that describes an output iterator.

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator**.

money_put::money_put

The constructor for objects of type `money_put`.

```
explicit money_put(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: the lifetime of the object is managed by the locales that contain it.
- 1: the lifetime of the object must be manually managed.
- > 1: these values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::facet**(`_Refs`).

money_put::put

Converts either number or a string to a character sequence that represents a monetary value.

```
iter_type put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    const string_type& val) const;

iter_type put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

Parameters

next

An iterator addressing the first element of the inserted string.

_Intl

A Boolean value indicating the type of currency symbol expected in the sequence: **true** if international, **false** if domestic.

_Iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill

A character which is used for spacing.

val

A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

Both member functions return `do_put(next, _Intl, _Iosbase, _Fill, val)`.

Example

```
// money_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

int main()
{
    std::locale loc( "german_germany" );
    std::basic_stringstream<char> psz;

    psz.imbue(loc);
    psz.flags(psz.flags() | std::ios_base::showbase); // force the printing of the currency symbol
    std::use_facet<std::money_put<char> >(loc).put(std::basic_ostream<char>::_Iter(psz.rdbuf()), true, psz, '
', 100012);
    if (psz.fail())
        std::cout << "money_put() FAILED" << std::endl;
    else
        std::cout << "money_put() = \"" << psz.rdbuf()->str() << "\"" << std::endl;
}
```

```
money_put() = "EUR1.000,12"
```

money_put::string_type

A type that describes a string containing characters of type `CharType` .

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of template class `basic_string` whose objects can store sequences of elements from the source sequence.

See also

[<locale>](#)

[facet Class](#)

[Thread Safety in the C++ Standard Library](#)

moneypunct Class

10/31/2018 • 14 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to describe the sequences of type *CharType* used to represent a monetary input field or a monetary output field. If the template parameter *Intl* is *true*, international conventions are observed.

Syntax

```
template <class CharType, bool Intl>
class moneypunct;
```

Parameters

CharType

The type used within a program to encode characters.

Intl

A flag specifying whether international conventions are to be observed.

Remarks

As with any locale facet, the static object *ID* has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

The const static object *intl* stores the value of the template parameter *Intl*.

Constructors

CONSTRUCTOR	DESCRIPTION
moneypunct	Constructor of objects of type <code>moneypunct</code> .

Typedefs

TYPE NAME	DESCRIPTION
char_type	A type that is used to describe a character used by a locale.
string_type	A type that describes a string containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
curr_symbol	Returns a locale-specific sequence of elements to use as a currency symbol.
decimal_point	Returns a locale-specific sequence of elements to use as a decimal point symbol.

MEMBER FUNCTION	DESCRIPTION
do_curr_symbol	A protected virtual member function that returns a locale-specific sequence of elements to use as a currency symbol.
do_decimal_point	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a decimal point symbol.
do_frac_digits	The protected virtual member function returns a locale-specific count of the number of digits to display to the right of any decimal point.
do_grouping	The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
do_neg_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs with negative amounts.
do_negative_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.
do_pos_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs with positive amounts.
do_positive_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a positive sign symbol.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a thousands separator symbol.
frac_digits	Returns a locale-specific count of the number of digits to display to the right of any decimal point.
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
neg_format	Returns a locale-specific rule for formatting outputs with negative amounts.
negative_sign	Returns a locale-specific sequence of elements to use as a negative sign symbol.
pos_format	Returns a locale-specific rule for formatting outputs with positive amounts.
positive_sign	Returns a locale-specific sequence of elements to use as a positive sign symbol.
thousands_sep	Returns a locale-specific sequence of elements to use as a thousands separator symbol.

MEMBER FUNCTION	DESCRIPTION
-----------------	-------------

Requirements

Header: <locale>

Namespace: std

moneypunct::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

moneypunct::curr_symbol

Returns a locale-specific sequence of elements to use as a currency symbol.

```
string_type curr_symbol() const;
```

Return Value

A string containing the currency symbol.

Remarks

The member function returns [do_curr_symbol](#).

Example

```
// moneypunct_curr_symbol.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const moneypunct< char, true > &mpunct = use_facet< moneypunct< char, true >>(loc);
    cout << loc.name( ) << " international currency symbol "<< mpunct.curr_symbol( ) << endl;

    const moneypunct< char, false> &mpunct2 = use_facet< moneypunct< char, false>>(loc);
    cout << loc.name( ) << " domestic currency symbol "<< mpunct2.curr_symbol( ) << endl;
};
```

moneypunct::decimal_point

Returns a locale-specific sequence of elements to use as a decimal point symbol.

```
CharType decimal_point() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Remarks

The member function returns [do_decimal_point](#).

Example

```
// moneypunct_decimal_pt.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc("german_germany");

    const moneypunct < char, true > &mpunct = use_facet
        < moneypunct < char, true > >(loc);
    cout << loc.name( ) << " international decimal point "
        << mpunct.decimal_point( ) << endl;

    const moneypunct < char, false> &mpunct2 = use_facet
        < moneypunct < char, false> >(loc);
    cout << loc.name( ) << " domestic decimal point "
        << mpunct2.decimal_point( ) << endl;
}
```

```
German_Germany.1252 international decimal point ,
German_Germany.1252 domestic decimal point ,
```

moneypunct::do_curr_symbol

A protected virtual member function that returns a locale-specific sequence of elements to use as a currency symbol.

```
virtual string_type do_curr_symbol() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for [curr_symbol](#), where the virtual member function is called by `curr_symbol`.

moneypunct::do_decimal_point

A protected virtual member function that returns a locale-specific sequence of elements to use as a decimal point symbol.

```
virtual CharType do_decimal_point() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for [decimal_point](#), where the virtual member function is called by `decimal_point`.

moneypunct::do_frac_digits

A protected virtual member function that returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
virtual int do_frac_digits() const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Example

See the example for [frac_digits](#), where the virtual member function is called by `frac_digits`.

moneypunct::do_grouping

A protected virtual member function that returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
virtual string do_grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Example

See the example for [grouping](#), where the virtual member function is called by `grouping`.

moneypunct::do_neg_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with negative amounts.

```
virtual pattern do_neg_format() const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a negative amount. Each of the four elements of `pattern::field` can have the values:

- `none` to match zero or more spaces or generate nothing.
- `sign` to match or generate a positive or negative sign.
- `space` to match zero or more spaces or generate a space.
- `symbol` to match or generate a currency symbol.
- `value` to match or generate a monetary value.

Components of a monetary output field are generated and components of a monetary input field are matched in the order in which these elements appear in `pattern::field`. Each of the values `sign`, `symbol`, `value`, and either `none` or `space` must appear exactly once. The value `none` must not appear first. The value `space` **must** not appear first or last. If `Intl` is true, the order is `symbol`, `sign`, `none`, then `value`.

The template version of `moneypunct` < **CharType**, **Intl** > returns `{ money_base::symbol, money_base::sign, money_base::value, money_base::none }`.

Example

See the example for [neg_format](#), where the virtual member function is called by `neg_format`.

moneypunct::do_negative_sign

A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.

```
virtual string_type do_negative_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a negative sign.

Example

See the example for [negative_sign](#), where the virtual member function is called by `negative_sign`.

moneypunct::do_pos_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with positive amounts.

```
virtual pattern do_pos_format() const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a positive amount. (It also determines how to match the components of a monetary input field.) The encoding is the same as for [do_neg_format](#).

The template version of `moneypunct` < **CharType**, **InputIterator** > returns `{ money_base::symbol, money_base::sign, money_base::value, money_base::none }`.

Example

See the example for [pos_format](#), where the virtual member function is called by `pos_format`.

moneypunct::do_positive_sign

A protected virtual member function that returns a locale-specific sequence of elements to use as a positive sign.

```
virtual string_type do_positive_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign.

Example

See the example for [positive_sign](#), where the virtual member function is called by `positive_sign`.

money_punct::do_thousands_sep

A protected virtual member function that returns a locale-specific element to use as a group separator to the left of any decimal point.

```
virtual CharType do_thousands_sep() const;
```

Return Value

A locale-specific element to use as a group separator to the left of any decimal point.

Example

See the example for [thousands_sep](#), where the virtual member function is called by `thousands_sep`.

money_punct::frac_digits

Returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
int frac_digits() const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Remarks

The member function returns [do_frac_digits](#).

Example

```

// moneypunct_frac_digits.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    for (unsigned int i = 0; i < mpunct.grouping( ).length( ); i++ )
    {
        cout << loc.name( ) << " international grouping:\n the "
            << i << "th group to the left of the radix character "
            << "is of size " << (int)(mpunct.grouping ( )[i])
            << endl;
    }
    cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    for (unsigned int i = 0; i < mpunct2.grouping( ).length( ); i++ )
    {
        cout << loc.name( ) << " domestic grouping:\n the "
            << i << "th group to the left of the radix character "
            << "is of size " << (int)(mpunct2.grouping ( )[i])
            << endl;
    }
    cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}

```

```

German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
to the right of the radix character: 2

German_Germany.1252 domestic grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
to the right of the radix character: 2

```

moneypunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns [do_grouping](#).

Example

```

// moneypunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >( loc );
    for (unsigned int i = 0; i <mpunct.grouping( ).length( ); i++ )
    {
        cout << loc.name( ) << " international grouping:\n the "
            << i <<"th group to the left of the radix character "
            << "is of size " << (int)(mpunct.grouping ( )[i])
            << endl;
    }
    cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >( loc );
    for (unsigned int i = 0; i <mpunct2.grouping( ).length( ); i++ )
    {
        cout << loc.name( ) << " domestic grouping:\n the "
            << i <<"th group to the left of the radix character "
            << "is of size " << (int)(mpunct2.grouping ( )[i])
            << endl;
    }
    cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}

```

```

German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
to the right of the radix character: 2

German_Germany.1252 domestic grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
to the right of the radix character: 2

```

moneypunct::moneypunct

Constructor of objects of type `moneypunct`.

```
explicit moneypunct(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with [locale::facet\(_ Refs\)](#).

moneypunct::neg_format

Returns a locale-specific rule for formatting outputs with negative amounts.

```
pattern neg_format() const;
```

Return Value

A locale-specific rule for formatting outputs with negative amounts.

Remarks

The member function returns [do_neg_format](#).

Example

```
// moneypunct_neg_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( ) {
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    cout << loc.name( ) << " international negative number format: "
        << mpunct.neg_format( ).field[0]
        << mpunct.neg_format( ).field[1]
        << mpunct.neg_format( ).field[2]
        << mpunct.neg_format( ).field[3] << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    cout << loc.name( ) << " domestic negative number format: "
        << mpunct2.neg_format( ).field[0]
        << mpunct2.neg_format( ).field[1]
        << mpunct2.neg_format( ).field[2]
        << mpunct2.neg_format( ).field[3] << endl;
}
```

moneypunct::negative_sign

Returns a locale-specific sequence of elements to use as a negative sign symbol.

```
string_type negative_sign() const;
```

Return Value

Returns a locale-specific sequence of elements to use as a negative sign symbol.

Remarks

The member function returns [do_negative_sign](#).

Example

```
// moneypunct_neg_sign.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    cout << loc.name( ) << " international negative sign: "
        << mpunct.negative_sign( ) << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    cout << loc.name( ) << " domestic negative sign: "
        << mpunct2.negative_sign( ) << endl;

    locale loc2( "French" );

    const moneypunct <char, true> &mpunct3 =
        use_facet <moneypunct <char, true> >(loc2);
    cout << loc2.name( ) << " international negative sign: "
        << mpunct3.negative_sign( ) << endl;

    const moneypunct <char, false> &mpunct4 =
        use_facet <moneypunct <char, false> >(loc2);
    cout << loc2.name( ) << " domestic negative sign: "
        << mpunct4.negative_sign( ) << endl;
};
```

```
German_Germany.1252 international negative sign: -
German_Germany.1252 domestic negative sign: -
French_France.1252 international negative sign: -
French_France.1252 domestic negative sign: -
```

moneypunct::pos_format

Returns a locale-specific rule for formatting outputs with positive amounts.

```
pattern pos_format() const;
```

Return Value

A locale-specific rule for formatting outputs with positive amounts.

Remarks

The member function returns [do_pos_format](#).

Example

```

// moneypunct_pos_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main() {
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    cout << loc.name( ) << " international positive number format: "
        << mpunct.pos_format( ).field[0]
        << mpunct.pos_format( ).field[1]
        << mpunct.pos_format( ).field[2]
        << mpunct.pos_format( ).field[3] << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    cout << loc.name( ) << " domestic positive number format: "
        << mpunct2.pos_format( ).field[0]
        << mpunct2.pos_format( ).field[1]
        << mpunct2.pos_format( ).field[2]
        << mpunct2.pos_format( ).field[3] << endl;
}

```

moneypunct::positive_sign

Returns a locale-specific sequence of elements to use as a positive sign symbol.

```
string_type positive_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign symbol.

Remarks

The member function returns [do_positive_sign](#).

Example

```

// moneypunct_pos_sign.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    cout << loc.name( ) << " international positive sign:"
        << mpunct.positive_sign( ) << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    cout << loc.name( ) << " domestic positive sign:"
        << mpunct2.positive_sign( ) << endl;

    locale loc2( "French" );

    const moneypunct <char, true> &mpunct3 =
        use_facet <moneypunct <char, true> >(loc2);
    cout << loc2.name( ) << " international positive sign:"
        << mpunct3.positive_sign( ) << endl;

    const moneypunct <char, false> &mpunct4 =
        use_facet <moneypunct <char, false> >(loc2);
    cout << loc2.name( ) << " domestic positive sign:"
        << mpunct4.positive_sign( ) << endl;
};

```

```

German_Germany.1252 international positive sign:
German_Germany.1252 domestic positive sign:
French_France.1252 international positive sign:
French_France.1252 domestic positive sign:

```

moneypunct::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of template class [basic_string](#) whose objects can store copies of the punctuation sequences.

moneypunct::thousands_sep

Returns a locale-specific sequence of elements to use as a thousands separator symbol.

```
CharType thousands_sep() const;
```

Return Value

A locale-specific sequence of elements to use as a thousands separator

Remarks

The member function returns [do_thousands_sep](#).

Example

```
// moneypunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const moneypunct <char, true> &mpunct =
        use_facet <moneypunct <char, true> >(loc);
    cout << loc.name( ) << " international thousands separator: "
        << mpunct.thousands_sep( ) << endl;

    const moneypunct <char, false> &mpunct2 =
        use_facet <moneypunct <char, false> >(loc);
    cout << loc.name( ) << " domestic thousands separator: "
        << mpunct2.thousands_sep( ) << endl << endl;

    locale loc2( "english_canada" );

    const moneypunct <char, true> &mpunct3 =
        use_facet <moneypunct <char, true> >(loc2);
    cout << loc2.name( ) << " international thousands separator: "
        << mpunct3.thousands_sep( ) << endl;

    const moneypunct <char, false> &mpunct4 =
        use_facet <moneypunct <char, false> >(loc2);
    cout << loc2.name( ) << " domestic thousands separator: "
        << mpunct4.thousands_sep( ) << endl;
}
```

```
German_Germany.1252 international thousands separator: .
German_Germany.1252 domestic thousands separator: .

English_Canada.1252 international thousands separator: ,
English_Canada.1252 domestic thousands separator: ,
```

See also

[<locale>](#)

[Thread Safety in the C++ Standard Library](#)

moneypunct_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A derived template class that describes an object that can serve as a `moneypunct` facet of a given locale, enabling the formatting monetary input field or monetary output fields.

Syntax

```
template <class CharType, bool Intl = false>
class moneypunct_byname : public moneypunct<CharType, Intl>
{
public:
    explicit moneypunct_byname(
        const char* _Locname,
        size_t _Refs = 0);

    explicit moneypunct_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual ~moneypunct_byname();

};
```

Remarks

Its behavior is determined by the named locale `_Locname`. Each constructor initializes its base object with `moneypunct<CharType, Intl>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

num_get Class

10/31/2018 • 11 minutes to read • [Edit Online](#)

A template class that describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to numeric values.

Syntax

```
template <class CharType, class InputIterator = istreambuf_iterator<CharType>>
class num_get : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

InputIterator

The type of iterator from which the numeric get functions read their input.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **`id`**.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>num_get</code>	The constructor for objects of type <code>num_get</code> that are used to extract numerical values from sequences.

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>iter_type</code>	A type that describes an input iterator.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>do_get</code>	A virtual function that is called to extracts a numerical or Boolean value from a character sequence.
<code>get</code>	Extracts a numerical or Boolean value from a character sequence.

Requirements

Header: <locale>

Namespace: std

num_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

num_get::do_get

A virtual function that is called to extracts a numerical or Boolean value from a character sequence.

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned short& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned int& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    long long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned long long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
```

```

        ios_base::iostate& _State,
        float& val) const;

    virtual iter_type do_get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        double& val) const;

    virtual iter_type do_get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        long double& val) const;

    virtual iter_type do_get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        void *& val) const;

    virtual iter_type do_get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        bool& val) const;

```

Parameters

first

The beginning of the range of characters from which to read the number.

last

The end of the range of characters from which to read the number.

_Iosbase

The [ios_base](#) whose flags are used by the conversion.

_State

The state to which failbit (see [ios_base::iostate](#)) is added upon failure.

val

The value that was read.

Return Value

The iterator after the value has been read.

Remarks

The first virtual protected member function,

```

    virtual iter_type do_get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        long& val) const;

```

matches sequential elements beginning at *first* in the sequence `[first, last)` until it has recognized a complete, nonempty integer input field. If successful, it converts this field to its equivalent value as type **long**, and stores the

result in *val*. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in *val* and sets `ios_base::failbit` in `state`. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `state`.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of **char** elements from a file. (Each such **char** element is assumed to map to an equivalent element of type `Elem` by a simple, one-to-one, mapping.) The equivalent scan conversion specification is determined as follows:

If `iosbase.ios_base::flags()` & `ios_base::basefield` == `ios_base::oct`, the conversion specification is `lo`.

If `iosbase.flags()` & `ios_base::basefield` == `ios_base::hex`, the conversion specification is `lx`.

If `iosbase.flags()` & `ios_base::basefield` == `0`, the conversion specification is `li`.

Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the locale facet `fac` returned by the call `use_facet<num_punct>(iosbase.ios_base::getloc())`. Specifically:

`fac.num_punct::grouping()` determines how digits are grouped to the left of any decimal point

`fac.num_punct::thousands_sep()` determines the sequence that separates groups of digits to the left of any decimal point.

If no instances of `fac.thousands_sep()` occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` are enforced and separators are removed before the scan conversion occurs.

The fourth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful it converts the numeric input field to a value of type **unsigned long** and stores that value in *val*.

The fifth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    long long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lld`. If successful it converts the numeric input field to a value of type **long long** and stores that value in *val*.

The sixth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned long long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `llu`. If successful it converts the numeric input field to a value of type **unsigned long long** and stores that value in *val*.

The seventh virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    float& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty floating-point input field.

`fac. numpunct::decimal_point()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The eighth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    double& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty floating-point input field.

`fac. numpunct::decimal_point()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The ninth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    long double& val) const;
```

behaves the same as the eighth, except that the equivalent scan conversion specifier is `Lf`.

The tenth virtual protected member function:

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The last (eleventh) virtual protected member function:

```
virtual iter_type do_get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    bool& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty Boolean input field. If successful it converts the Boolean input field to a value of type **bool** and stores that value in *val*.

A Boolean input field takes one of two forms. If `iosbase.flags() & ios_base::boolalpha` is false, it is the same as an integer input field, except that the converted value must be either 0 (for false) or 1 (for true). Otherwise, the sequence must match either `fac. numpunct::falsename()` (for false), or `fac. numpunct::truename()` (for true).

Example

See the example for [get](#), where the virtual member function is called by `do_get`.

num_get::get

Extracts a numerical or Boolean value from a character sequence.

```
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    bool& val) const;  
  
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    unsigned short& val) const;  
  
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    unsigned int& val) const;  
  
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    long& val) const;  
  
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,  
    unsigned long& val) const;  
  
iter_type get(  
    iter_type first,  
    iter_type last,  
    ios_base& _Iosbase,  
    ios_base::iostate& _State,
```



```

        long long& val) const;

    iter_type get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        unsigned long long& val) const;

    iter_type get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        float& val) const;

    iter_type get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        double& val) const;

    iter_type get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        long double& val) const;

    iter_type get(
        iter_type first,
        iter_type last,
        ios_base& _Iosbase,
        ios_base::iostate& _State,
        void *& val) const;

```

Parameters

first

The beginning of the range of characters from which to read the number.

last

The end of the range of characters from which to read the number.

_Iosbase

The [ios_base](#) whose flags are used by the conversion.

_State

The state to which failbit (see [ios_base::iostate](#)) is added upon failure.

val

The value that was read.

Return Value

The iterator after the value has been read.

Remarks

All member functions return [do_get](#)([first](#), [last](#), [_Iosbase](#), [_State](#), [val](#)).

The first virtual protected member function tries to match sequential elements beginning at [first](#) in the sequence [[first](#), [last](#)) until it has recognized a complete, nonempty integer input field. If successful, it converts this field to its equivalent value as type **long** and stores the result in [val](#). It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in [val](#) and sets [ios_base::failbit](#) in [_State](#).

It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in `_State`.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of **char** elements from a file. Each such **char** element is assumed to map to an equivalent element of type `charType` by a simple, one-to-one mapping. The equivalent scan conversion specification is determined as follows:

- If `iosbase.flags & ios_base::basefield == ios_base::oct`, the conversion specification is `lo`.
- If `iosbase.flags & ios_base::basefield == ios_base::hex`, the conversion specification is `lx`.
- If `iosbase.flags & ios_base::basefield == 0`, the conversion specification is `li`.
- Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the `locale facet` `fac` returned by the call `use_facet < num_punct > (iosbase.getloc())`. Specifically:

- **fac.grouping** determines how digits are grouped to the left of any decimal point.
- **fac.thousands_sep** determines the sequence that separates groups of digits to the left of any decimal point.

If no instances of **fac.thousands_sep** occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by **fac.grouping** is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful, it converts the numeric input field to a value of type **unsigned long** and stores that value in *val*.

The third virtual protected member function:

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    double& val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty floating-point input field. **fac.decimal_point** determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The fourth virtual protected member function:

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    long double& val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is `Lf`.

The fifth virtual protected member function:

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The sixth virtual protected member function:

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& _Iosbase,
    ios_base::iostate& _State,
    bool& val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty boolean input field. If successful it converts the Boolean input field to a value of type **bool** and stores that value in *val*.

A boolean input field takes one of two forms. If **iosbase.flags** & `ios_base::boolalpha` is **false**, it is the same as an integer input field, except that the converted value must be either 0 (for **false**) or 1 (for **true**). Otherwise, the sequence must match either **fac.falsename** (for **false**), or **fac.true name** (for **true**).

Example

```
// num_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    basic_stringstream<char> psz, psz2;
    psz << "-1000,56";

    ios_base::iostate st = 0;
    long double fVal;
    cout << use_facet <num_punct <char> >(loc).thousands_sep( ) << endl;

    psz.imbue( loc );
    use_facet <num_get <char> >
    (loc).get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
        basic_istream<char>::_Iter(0), psz, st, fVal );

    if ( st & ios_base::failbit )
        cout << "money_get( ) FAILED" << endl;
    else
        cout << "money_get( ) = " << fVal << endl;
}
```

num_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `InputIterator`.

num_get::num_get

The constructor for objects of type `num_get` that are used to extract numerical values from sequences.

```
explicit num_get(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with `locale::facet(_Refs)`.

See also

[<locale>](#)

[facet Class](#)

[Thread Safety in the C++ Standard Library](#)

num_put Class

11/8/2018 • 8 minutes to read • [Edit Online](#)

A template class that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type `CharType`.

Syntax

```
template <class CharType,  
        class OutputIterator = ostreambuf_iterator<CharType>>  
class num_put : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

OutputIterator

The type of iterator to which the numeric put functions write their output.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>num_put</code>	The constructor for objects of type <code>num_put</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>iter_type</code>	A type that describes an output iterator.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>do_put</code>	A virtual function that is called to convert a number into a sequence of <code>CharType</code> s that represents the number formatted for a given locale.
<code>put</code>	Converts a number into a sequence of <code>CharType</code> s which represents the number formatted for a given locale.

Requirements

Header: <locale>

Namespace: std

num_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

num_put::do_put

A virtual function that is called to convert a number into a sequence of `CharType`s that represents the number formatted for a given locale.

```

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    bool val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    unsigned long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    double val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long double val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const void* val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const long long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const unsigned long long val) const;

```

Parameters

next

An iterator addressing the first element of the inserted string.

_Iosbase

Specified the stream which contains locale with the numpunct facet used to punctuate the output and flags for formatting the output.

_Fill

A character that is used for spacing.

val

The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *next* to produce an integer output field from the value of *val*. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions for generating a series of **char** elements to a file. Each such char element is assumed to map to an equivalent element of type `CharType` by a simple, one-to-one mapping. Where a print function pads a field with either spaces or the digit 0, however, `do_put` instead uses `fill`. The equivalent print conversion specification is determined as follows:

- If `iosbase.flags & ios_base::basefield == ios_base::oct`, the conversion specification is `lo`.
- If `iosbase.flags & ios_base::basefield == ios_base::hex`, the conversion specification is `lx`.
- Otherwise, the conversion specification is `ld`.

If `iosbase.width` is nonzero, a field width of this value is prepended. The function then calls `iosbase.width(0)` to reset the field width to zero.

Padding occurs only if the minimum number of elements *N* required to specify the output field is less than `iosbase.width`. Such padding consists of a sequence of *N* - **width** copies of `fill`. Padding then occurs as follows:

- If `iosbase.flags & ios_base::adjustfield == ios_base::left`, the flag - is prepended. (Padding occurs after the generated text.)
- If `iosbase.flags & ios_base::adjustfield == ios_base::internal`, the flag 0 is prepended. (For a numeric output field, padding occurs where the print functions pad with 0.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:

- If `iosbase.flags & ios_base::showpos` is nonzero, the flag + is prepended to the conversion specification.
- If `iosbase.flags & ios_base::showbase` is nonzero, the flag # is prepended to the conversion specification.

The format of an integer output field is further determined by the `locale facet` returned by the call `use_facet < num_punct < Elem > (iosbase.getloc)`. Specifically:

- **fac.grouping** determines how digits are grouped to the left of any decimal point
- **fac.thousands_sep** determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by **fac.grouping** (its first element has the value `CHAR_MAX`), then no instances of **fac.thousands_sep** are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    unsigned long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`.

The third virtual protected member function:


```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    double val) const;
```

behaves the same as the first, except that it produces a floating-point output field from the value of **val**. **fac**. [decimal_point](#) determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specification is determined as follows:

- If **iosbase.flags** & `ios_base::floatfield` == `ios_base::fixed`, the conversion specification is `lf`.
- If **iosbase.flags** & `ios_base::floatfield` == `ios_base::scientific`, the conversion specification is `le`. If **iosbase.flags** & `ios_base::uppercase` is nonzero, `e` is replaced with `E`.
- Otherwise, the conversion specification is **lg**. If **iosbase.flags** & `ios_base::uppercase` is nonzero, `g` is replaced with `G`.

If **iosbase.flags** & `ios_base::fixed` is nonzero or if **iosbase.precision** is greater than zero, a precision with the value **iosbase.precision** is prepended to the conversion specification. Any padding behaves the same as for an integer output field. The padding character is **fill**. Finally:

- If **iosbase.flags** & `ios_base::showpos` is nonzero, the flag `+` is prepended to the conversion specification.
- If **iosbase.flags** & `ios_base::showpoint` is nonzero, the flag `#` is prepended to the conversion specification.

The fourth virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

behaves the same the third, except that the qualifier `l` in the conversion specification is replaced with `L`.

The fifth virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    const void* val) const;
```

behaves the same the first, except that the conversion specification is `p`, plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    bool val) const;
```

behaves the same as the first, except that it generates a Boolean output field from *val*.

A Boolean output field takes one of two forms. If `iosbase.flags` & `ios_base::boolalpha` is **false**, the member function returns `do_put(_Next, _Iosbase, _Fill, (long)val)`, which typically produces a generated sequence of either 0 (for **false**) or 1 (for **true**). Otherwise, the generated sequence is either *fac.falsename* (for **false**), or

`fac.truename` (for **true**).

The seventh virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& iosbase,
    Elem fill,
    long long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lld`.

The eighth virtual protected member function:

```
virtual iter_type do_put(iter_type next,
    ios_base& iosbase,
    Elem fill,
    unsigned long long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `llu`.

Example

See the example for `put`, which calls `do_put`.

num_put::iter_type

A type that describes an output iterator.

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator**.

num_put::num_put

The constructor for objects of type `num_put`.

```
explicit num_put(size_t _Refs = 0);
```

Parameters

`_Refs`

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the `_Refs` parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with `locale::facet(_Refs)`.

num_put::put

Converts a number into a sequence of `CharType`s that represents the number formatted for a given locale.

```
iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    bool val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    unsigned long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    Long long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    Unsigned long long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    double val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long double val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const void* val) const;
```

Parameters

dest

An iterator addressing the first element of the inserted string.

_Iosbase

Specified the stream that contains locale with the `numput` facet used to punctuate the output and flags for formatting the output.

_Fill

A character that is used for spacing.

val

The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

All member functions return `do_put(next, _Iosbase, _Fill, val)`.

Example

```
// num_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );
    basic_stringstream<char> psz2;
    ios_base::iostate st = 0;
    long double fVal;
    cout << "The thousands separator is: "
         << use_facet < numpunct <char> >(loc).thousands_sep( )
         << endl;

    psz2.imbue( loc );
    use_facet < num_put < char > >
        ( loc ).put(basic_ostream<char>::_Iter(psz2.rdbuf( )),
                psz2, ' ', fVal=1000.67);

    if ( st & ios_base::failbit )
        cout << "num_put( ) FAILED" << endl;
    else
        cout << "num_put( ) = " << psz2.rdbuf( )->str( ) << endl;
}
```

```
The thousands separator is: .
num_put( ) = 1.000,67
```

See also

[<locale>](#)

[facet Class](#)

[Thread Safety in the C++ Standard Library](#)

numpunct Class

10/31/2018 • 7 minutes to read • [Edit Online](#)

A template class that describes an object that can serve as a local facet to describe the sequences of type `CharType` used to represent information about the formatting and punctuation of numeric and Boolean expressions.

Syntax

```
template <class CharType>
class numpunct : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters in a locale.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>numpunct</code>	The constructor for objects of type <code>numpunct</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>string_type</code>	A type that describes a string containing characters of type <code>CharType</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>decimal_point</code>	Returns a locale-specific element to use as a decimal point.
<code>do_decimal_point</code>	A protected virtual member function that is called to return a locale-specific element to use as a decimal point.
<code>do_falsename</code>	A protected virtual member function that is called to return a string to use as a text representation of the value false .

MEMBER FUNCTION	DESCRIPTION
do_grouping	A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.
do_truename	A protected virtual member function that is called to return a string to use as a text representation of the value true .
falsename	Returns a string to use as a text representation of the value false .
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
thousands_sep	Returns a locale-specific element to use as a thousands separator.
truename	Returns a string to use as a text representation of the value true .

Requirements

Header: <locale>

Namespace: std

num_punct::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

num_punct::decimal_point

Returns a locale-specific element to use as a decimal point.

```
CharType decimal_point() const;
```

Return Value

A locale-specific element to use as a decimal point.

Remarks

The member function returns [do_decimal_point](#).

Example

```
// numpunct_decimal_point.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const numpunct <char> &npunct =
        use_facet <numpunct <char> >( loc);
    cout << loc.name( ) << " decimal point "<<
        npunct.decimal_point( ) << endl;
    cout << loc.name( ) << " thousands separator "
        << npunct.thousands_sep( ) << endl;
};
```

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

numpunct::do_decimal_point

A protected virtual member function that is called to return a locale-specific element to use as a decimal point.

```
virtual CharType do_decimal_point() const;
```

Return Value

A locale-specific element to use as a decimal point.

Example

See the example for [decimal_point](#), where the virtual member function is called by `decimal_point`.

numpunct::do_falsename

The protected virtual member function returns a sequence to use as a text representation of the value **false**.

```
virtual string_type do_falsename() const;
```

Return Value

A string containing a sequence to use as a text representation of the value **false**.

Remarks

The member function returns the string "false" to represent the value **false** in all locales.

Example

See the example for [falsename](#), where the virtual member function is called by `falsename`.

numpunct::do_grouping

A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
virtual string do_grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for **lconv::grouping**.

Example

See the example for [grouping](#), where the virtual member function is called by `grouping`.

numpunct::do_thousands_sep

A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.

```
virtual CharType do_thousands_sep() const;
```

Return Value

Returns a locale-specific element to use as a thousands separator.

Remarks

The protected virtual member function returns a locale-specific element of type `CharType` to use as a group separator to the left of any decimal point.

Example

See the example for [thousands_sep](#), where the virtual member function is called by `thousands_sep`.

numpunct::do_truename

A protected virtual member function that is called to return a string to use as a text representation of the value **true**.

```
virtual string_type do_truename() const;
```

Remarks

A string to use as a text representation of the value **true**.

All locales return a string "true" to represent the value **true**.

Example

See the example for [truename](#), where the virtual member function is called by `truename`.

numpunct::falsename

Returns a string to use as a text representation of the value **false**.

```
string_type falsename() const;
```

Return Value

A string containing a sequence of `CharType`s to use as a text representation of the value **false**.

Remarks

The member function returns the string "false" to represent the value **false** in all locales.

The member function returns [do_falsename](#).

Example

```
// numpunct_falsename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "English" );

    const numpunct<char> &npunct = use_facet<numpunct<char>>( loc );
    cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
    cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

    locale loc2( "French" );
    const numpunct<char> &npunct2 = use_facet<numpunct<char>>(loc2);
    cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
    cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}
```

```
English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false
```

numpunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
string grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns [do_grouping](#).

Example

```
// numpunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany");

    const numpunct <char> &npunct =
        use_facet < numpunct <char> >( loc );
    for (unsigned int i = 0; i < npunct.grouping( ).length( ); i++)
    {
        cout << loc.name( ) << " international grouping:\n the "
            << i <<"th group to the left of the radix character "
            << "is of size " << (int)(npunct.grouping ( )[i])
            << endl;
    }
}
```

German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3

numpunct::numpunct

The constructor for objects of type `numpunct`.

```
explicit numpunct(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with `locale::facet`(`_Refs`).

numpunct::string_type

A type that describes a string containing characters of type **CharType**.

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of template class `basic_string` whose objects can store copies of the punctuation sequences.

numpunct::thousands_sep

Returns a locale-specific element to use as a thousands separator.

```
CharType thousands_sep() const;
```

Return Value

A locale-specific element to use as a thousands separator.

Remarks

The member function returns [do_thousands_sep](#).

Example

```
// numpunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "german_germany" );

    const numpunct <char> &npunct =
        use_facet < numpunct < char > >( loc );
    cout << loc.name( ) << " decimal point "<<
        npunct.decimal_point( ) << endl;
    cout << loc.name( ) << " thousands separator "
        << npunct.thousands_sep( ) << endl;
};
```

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

numpunct::truename

Returns a string to use as a text representation of the value **true**.

```
string_type falsename() const;
```

Return Value

A string to use as a text representation of the value **true**.

Remarks

The member function returns [do_truename](#).

All locales return a string "true" to represent the value **true**.

Example

```

// numpunct_truename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
    locale loc( "English" );

    const numpunct < char> &npunct = use_facet <numpunct <char> >( loc );
    cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
    cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

    locale loc2("French");
    const numpunct <char> &npunct2 = use_facet <numpunct <char> >( loc2 );
    cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
    cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}

```

```

English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false

```

See also

[<locale>](#)

[facet Class](#)

[Thread Safety in the C++ Standard Library](#)

numpunct_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The derived template class describes an object that can serve as a `numpunct` facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.

Syntax

```
template <class CharType>
class numpunct_byname : public numpunct<Elem> {
public:
    explicit numpunct_byname(
        const char* _Locname,
        size_t _Refs = 0);

    explicit numpunct_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual ~numpunct_byname();

};
```

Remarks

Its behavior is determined by the [named](#) locale `_Locname`. The constructor initializes its base object with `numpunct<CharType>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

time_base Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as a base class for facets of template class `time_get`, defining just the enumerated type `dateorder` and several constants of this type.

Syntax

```
class time_base : public locale::facet {
public:
    enum dateorder {
        no_order,
        dmy,
        mdy,
        ymd,
        ydm
    };
    time_base(size_t _Refs = 0)
    ~time_base();
};
```

Remarks

Each constant characterizes a different way to order the components of a date. The constants are:

- `no_order` specifies no particular order.
- `dmy` specifies the order day, month, then year, as in 2 December 1979.
- `mdy` specifies the order month, day, then year, as in December 2, 1979.
- `ymd` specifies the order year, month, then day, as in 1979/12/2.
- `ydm` specifies the order year, day, then month, as in 1979: 2 Dec.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

time_get Class

10/31/2018 • 21 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to time values.

Syntax

```
template <class CharType,  
         class InputIterator = istreambuf_iterator<CharType>>  
class time_get : public time_base;
```

Parameters

CharType

The type used within a program to encode characters.

InputIterator

The iterator from which the time values are read.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **`id`**.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>time_get</code>	The constructor for objects of type <code>time_get</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>iter_type</code>	A type that describes an input iterator.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>date_order</code>	Returns the date order used by a facet.
<code>do_date_order</code>	A protected virtual member function that is called to return the date order used by a facet.
<code>do_get</code>	Reads and converts character data to a time value.

MEMBER FUNCTION	DESCRIPTION
do_get_date	A protected virtual member function that is called to parse a string as the date produced by the <code>x</code> specifier for <code>strptime</code> .
do_get_monthname	A protected virtual member function that is called to parse a string as the name of the month.
do_get_time	A protected virtual member function that is called to parse a string as the date produced by the <code>x</code> specifier for <code>strptime</code> .
do_get_weekday	A protected virtual member function that is called to parse a string as the name of the day of the week.
do_get_year	A protected virtual member function that is called to parse a string as the name of the year.
get	Reads from a source of character data and converts that data to a time that is stored in a time struct.
get_date	Parses a string as the date produced by the <code>x</code> specifier for <code>strptime</code> .
get_monthname	Parses a string as the name of the month.
get_time	Parses a string as the date produced by the <code>x</code> specifier for <code>strptime</code> .
get_weekday	Parses a string as the name of the day of the week.
get_year	Parses a string as the name of the year.

Requirements

Header: <locale>

Namespace: std

time_get::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

time_get::date_order

Returns the date order used by a facet.


```
dateorder date_order() const;
```

Return Value

The date order used by a facet.

Remarks

The member function returns [do_date_order](#).

Example

```
// time_get_date_order.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
void po( char *p )
{
    locale loc( p );

    time_get <char>::dateorder order = use_facet <time_get <char> >( loc ).date_order ( );
    cout << loc.name( );
    switch (order){
        case time_base::dmy: cout << "(day, month, year)" << endl;
            break;
        case time_base::mdy: cout << "(month, day, year)" << endl;
            break;
        case time_base::ydm: cout << "(year, day, month)" << endl;
            break;
        case time_base::ymd: cout << "(year, month, day)"<< endl;
            break;
        case time_base::no_order: cout << "(no_order)"<< endl;
            break;
    }
}

int main( )
{
    po( "C" );
    po( "german" );
    po( "English_Britain" );
}
```

```
C(month, day, year)
German_Germany.1252(day, month, year)
English_United Kingdom.1252(day, month, year)
```

time_get::do_date_order

A protected virtual member function that is called to return the date order used by a facet.

```
virtual dateorder do_date_order() const;
```

Return Value

The date order used by a facet.

Remarks

The virtual protected member function returns a value of type **time_base::dateorder**, which describes the order in which date components are matched by [do_get_date](#). In this implementation, the value is **time_base::mdy**, corresponding to dates of the form December 2, 1979.

Example

See the example for [date_order](#), which calls `do_date_order`.

time_get::do_get

Reads and converts character data to a time value. Accepts one conversion specifier and modifier.

```
virtual iter_type
do_get(
iter_type first,
iter_type last,
ios_base& iosbase,
ios_base::iostate& state,
tm* ptm,
char fmt,
char mod) const;
```

Parameters

first

An Input iterator that indicates the start of the sequence to convert.

last

An Input iterator that indicates the end of the sequence.

iosbase

A stream object.

state

A field in iosbase where appropriate bitmask elements are set to indicate errors.

ptm

A pointer to the time structure where the time is to be stored.

fmt

A conversion specifier character.

mod

An optional modifier character.

Return Value

Returns an iterator that designates the first unconverted element. A conversion failure sets `ios_base::failbit` in `state` and returns *first*.

Remarks

The virtual member function converts and skips one or more input elements in the range [`first`, `last`) to determine the values stored in one or more members of `*pt`. A conversion failure sets `ios_base::failbit` in `state` and returns *first*. Otherwise, the function returns an iterator designating the first unconverted element.

The conversion specifiers are:

`'a'` or `'A'` -- behaves the same as [time_get::get_weekday](#).

`'b'`, `'B'`, or `'h'` -- behaves the same as [time_get::get_monthname](#).

'c' -- behaves the same as "%b %d %H : %M : %S %Y" .

'C' -- converts a decimal input field in the range [0, 99] to the value `val` and stores `val * 100 - 1900` in `pt-&tm_year` .

'd' or 'e' -- converts a decimal input field in the range [1, 31] and stores its value in `pt-&tm_mday` .

'D' -- behaves the same as "%m / %d / %y" .

'H' -- converts a decimal input field in the range [0, 23] and stores its value in `pt-&tm_hour` .

'I' -- converts a decimal input field in the range [0, 11] and stores its value in `pt-&tm_hour` .

'j' -- converts a decimal input field in the range [1, 366] and stores its value in `pt-&tm_yday` .

'm' -- converts a decimal input field in the range [1, 12] to the value `val` and stores `val - 1` in and stores its value in `pt-&tm_mon` .

'M' -- converts a decimal input field in the range [0, 59] and stores its value in `pt-&tm_min` .

'n' or 't' -- behaves the same as " " .

'p' -- converts "AM" or "am" to zero and "PM" or "PM" to 12 and adds this value to `pt-&tm_hour` .

'r' -- behaves the same as "%I : %M : %S %p" .

'R' -- behaves the same as "%H %M" .

's' -- converts a decimal input field in the range [0, 59] and stores its value in `pt-&tm_sec` .

'T' or 'X' -- behaves the same as "%H : %M : S" .

'U' -- converts a decimal input field in the range [0, 53] and stores its value in `pt-&tm_yday` .

'w' -- converts a decimal input field in the range [0, 6] and stores its value in `pt-&tm_wday` .

'W' -- converts a decimal input field in the range [0, 53] and stores its value in `pt-&tm_yday` .

'x' -- behaves the same as "%d / %m / %y" .

'y' -- converts a decimal input field in the range [0, 99] to the value `val` and stores `val < 69 val + 100 : val` in `pt-&tm_year` .

'Y' -- behaves the same as [time_get::get_year](#) .

Any other conversion specifier sets `ios_base::failbit` in `state` and returns. In this implementation, any modifier has no effect.

time_get::do_get_date

A protected virtual member function that is called to parse a string as the date produced by the `x` specifier for `strptime` .

```
virtual iter_type do_get_date(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty date input field. If successful, it converts this field to its equivalent value as the components **tm::tm_mon**, **tm::tm_day**, and **tm::tm_year**, and stores the results in `ptm->tm_mon`, `ptm->tm_day`, and `ptm->tm_year`, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets `iosbase::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The format for the date input field is locale dependent. For the default locale, the date input field has the form MMM DD, YYYY, where:

- MMM is matched by calling [get_monthname](#), giving the month.
- DD is a sequence of decimal digits whose corresponding numeric value must be in the range [1, 31], giving the day of the month.
- YYYY is matched by calling [get_year](#), giving the year.

The literal spaces and commas must match corresponding elements in the input sequence.

Example

See the example for [get_date](#), which calls `do_get_date`.

time_get::do_get_monthname

A protected virtual member function that is called to parse a string as the name of the month.

```
virtual iter_type do_get_monthname(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

Unused.

state

An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty month input field. If successful, it converts this field to its equivalent value as the component **tm::tm_mon**, and stores the result in `ptm->tm_mon`. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as Jan, January, Feb, February, and so on. The converted value is the number of months since January.

Example

See the example for [get_monthname](#), which calls `do_get_monthname`.

time_get::do_get_time

A protected virtual member function that is called to parse a string as the date produced by the *X* specifier for `strftime`.

```
virtual iter_type do_get_time(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

Unused.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty time input field. If successful, it converts this field to its equivalent value as the components `tm::tm_hour`, `tm::tm_min`, and `tm::tm_sec`, and stores the results in `ptm->tm_hour`, `ptm->tm_min`, and `ptm->tm_sec`, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

In this implementation, the time input field has the form HH:MM:SS, where:

- HH is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 24), giving the hour of the day.
- MM is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the minutes past the hour.
- SS is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the seconds past the minute.

The literal colons must match corresponding elements in the input sequence.

Example

See the example for [get_time](#), which calls `do_get_time`.

time_get::do_get_weekday

A protected virtual member function that is called to parse a string as the name of the day of the week.

```
virtual iter_type do_get_weekday(iter_type first,
                                iter_type last,
                                ios_base& iosbase,
                                ios_base::iostate& state,
                                tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *first* in the sequence [`first`, `last`) until it has recognized a complete, nonempty weekday input field. If successful, it converts this field

to its equivalent value as the component **tm::tm_wday**, and stores the result in `ptm->tm_wday`. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid weekday input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as Sun, Sunday, Mon, Monday, and so on. The converted value is the number of days since Sunday.

Example

See the example for [get_weekday](#), which calls `do_get_weekday`.

time_get::do_get_year

A protected virtual member function that is called to parse a string as the name of the year.

```
virtual iter_type do_get_year(iter_type first,
                             iter_type last,
                             ios_base& iosbase,
                             ios_base::iostate& state,
                             tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *first* in the sequence [`first`, `last`) until it has recognized a complete, nonempty year input field. If successful, it converts this field to its equivalent value as the component **tm::tm_year**, and stores the result in `ptm->tm_year`. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range [1900, 2036). The stored value is this value minus 1900. In this implementation, values in the range [69, 136) represent the range of years [1969, 2036). Values in the range [0, 69) are also permissible, but may represent either the range of years [1900, 1969) or [2000, 2069), depending on the specific translation environment.

Example

See the example for [get_year](#), which calls `do_get_year`.

time_get::get

Reads from a source of character data and converts that data to a time that is stored in a time struct. The first function accepts one conversion specifier and modifier, the second accepts several.

```
iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm,
    char fmt,
    char mod) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm,
    char_type* fmt_first,
    char_type* fmt_last) const;
```

Parameters

first

Input iterator that indicates where the sequence to be converted starts.

last

Input iterator that indicates the end of the sequence to be converted.

iosbase

The stream.

state

The appropriate bitmask elements are set for the stream state to indicate errors.

ptm

Pointer to the time structure where the time is to be stored.

fmt

A conversion specifier character.

mod

An optional modifier character.

fmt_first

Points to where the format directives start.

fmt_last

Points to the end of the format directives.

Return Value

Returns an iterator to the first character after the data that was used to assign the time struct `*ptm`.

Remarks

The first member function returns `do_get(first, last, iosbase, state, ptm, fmt, mod)`.

The second member function calls `do_get` under the control of the format delimited by `[fmt_first, fmt_last)`. It treats the format as a sequence of fields, each of which determines the conversion of zero or more input elements

delimited by `[first, last)`. It returns an iterator designating the first unconverted element. There are three kinds of fields:

A per cent (%) in the format, followed by an optional modifier *mod* in the set [EOQ#], followed by a conversion specifier *fmt*, replaces *first* with the value returned by `do_get(first, last, iosbase, state, ptm, fmt, mod)`. A conversion failure sets `ios_base::failbit` in *state* and returns.

A whitespace element in the format skips past zero or more input whitespace elements.

Any other element in the format must match the next input element, which is skipped. A match failure sets `ios_base::failbit` in *state* and returns.

time_get::get_date

Parses a string as the date produced by the *x* specifier for `strftime`.

```
iter_type get_date(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns `do_get_date(first, last, iosbase, state, ptm)`.

Note that months are counted from 0 to 11.

Example

```

// time_get_get_date.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc;
    basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset(&t, 0, sizeof(struct tm));

    pszGetF << "July 4, 2000";
    pszGetF.imbue( loc );
    basic_istream<char>::_Iter i = use_facet <time_get<char> >
    (loc).get_date(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
        basic_istream<char>::_Iter(0), pszGetF, st, &t);

    if ( st & ios_base::failbit )
        cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
    else

        cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
        << "\ntm_sec: " << t.tm_sec
        << "\ntm_min: " << t.tm_min
        << "\ntm_hour: " << t.tm_hour
        << "\ntm_mday: " << t.tm_mday
        << "\ntm_mon: " << t.tm_mon
        << "\ntm_year: " << t.tm_year
        << "\ntm_wday: " << t.tm_wday
        << "\ntm_yday: " << t.tm_yday
        << "\ntm_isdst: " << t.tm_isdst
        << endl;
}

```

```

time_get(July 4, 2000) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 4
tm_mon: 6
tm_year: 100
tm_wday: 0
tm_yday: 0
tm_isdst: 0

```

time_get::get_monthname

Parses a string as the name of the month.

```

iter_type get_monthname(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;

```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

Unused.

state

An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns `do_get_monthname`(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

```
// time_get_get_monthname.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc ( "French" );
    basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    pszGetF << "juillet";
    pszGetF.imbue( loc );
    basic_istream<char>::_Iter i = use_facet <time_get <char> >
    (loc).get_monthname(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
        basic_istream<char>::_Iter(0), pszGetF, st, &t);

    if (st & ios_base::failbit)
        cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
    else

        cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
        << "\ntm_sec: " << t.tm_sec
        << "\ntm_min: " << t.tm_min
        << "\ntm_hour: " << t.tm_hour
        << "\ntm_mday: " << t.tm_mday
        << "\ntm_mon: " << t.tm_mon
        << "\ntm_year: " << t.tm_year
        << "\ntm_wday: " << t.tm_wday
        << "\ntm_yday: " << t.tm_yday
        << "\ntm_isdst: " << t.tm_isdst
        << endl;
}
```

```
time_get(juillet) =  
tm_sec: 0  
tm_min: 0  
tm_hour: 0  
tm_mday: 0  
tm_mon: 6  
tm_year: 0  
tm_wday: 0  
tm_yday: 0  
tm_isdst: 0
```

time_get::get_time

Parses a string as the date produced by the *X* specifier for `strftime`.

```
iter_type get_time(iter_type first,  
    iter_type last,  
    ios_base& iosbase,  
    ios_base::iostate& state,  
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

Unused.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns `do_get_time`(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

```

// time_get_get_time.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc;
    basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    pszGetF << "11:13:20";
    pszGetF.imbue( loc );
    basic_istream<char>::_Iter i = use_facet
        <time_get<char>>
        (loc).get_time(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);

    if (st & ios_base::failbit)
        cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
    else

        cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
            << "\ntm_sec: " << t.tm_sec
            << "\ntm_min: " << t.tm_min
            << "\ntm_hour: " << t.tm_hour
            << endl;
}

```

```

time_get::get_time(11:13:20) =
tm_sec: 20
tm_min: 13
tm_hour: 11

```

time_get::get_weekday

Parses a string as the name of the day of the week.

```

iter_type get_weekday(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;

```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns `do_get_weekday(first, last, iosbase, state, ptm)`.

Example

```
// time_get_get_weekday.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc ( "French" );
    basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    pszGetF << "mercredi";
    pszGetF.imbue(loc);
    basic_istream<char>::_Iter i = use_facet
        <time_get<char> >
        (loc).get_weekday(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);

    if (st & ios_base::failbit)
        cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
    else

        cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
            << "\ntm_wday: " << t.tm_wday
            << endl;
}
```

```
time_get::get_time(mercredi) =
tm_wday: 3
```

time_get::get_year

Parses a string as the name of the year.

```
iter_type get_year(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first

Input iterator addressing the beginning of the sequence to be converted.

last

Input iterator addressing the end of the sequence to be converted.

iosbase

A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state

Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm

A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns `do_get_year(first, last, iosbase, state, ptm)`.

Example

```
// time_get_get_year.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc;
    basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    pszGetF << "1928";

    pszGetF.imbue( loc );
    basic_istream<char>::_Iter i = use_facet
        <time_get<char> >
        (loc).get_year(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);

    if (st & ios_base::failbit)
        cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
    else

        cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") ="
            << "\ntm_year: " << t.tm_year
            << endl;
}
```

```
time_get::get_year(1928) =
tm_year: 28
```

time_get::iter_type

A type that describes an input iterator.

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

time_get::time_get

The constructor for objects of type `time_get` .

```
explicit time_get(size_t refs = 0);
```

Parameters

refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::facet**(`refs`).

See also

[<locale>](#)

[time_base Class](#)

[Thread Safety in the C++ Standard Library](#)

time_get_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The derived template class describes an object that can serve as a locale facet of type `time_get<CharType, InputIterator>`.

Syntax

```
template <class Elem, class InputIterator =  
    istreambuf_iterator<CharType, char_traits<CharType>>>  
class time_get_byname : public time_get<CharType, InputIterator>  
{  
public:  
    explicit time_get_byname(  
        const char* _Locname,  
        size_t _Refs = 0);  
  
    explicit time_get_byname(  
        const string& _Locname,  
        size_t _Refs = 0);  
  
protected:  
    virtual ~time_get_byname()  
};
```

Parameters

_Locname

A named locale.

_Refs

An initial reference count.

Requirements

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with `time_get<CharType, InputIterator>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

time_put Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

The template class describes an object that can serve as a locale facet to control conversions of time values to sequences of type `CharType`.

Syntax

```
template <class CharType,  
         class OutputIterator = ostreambuf_iterator<CharType>>  
class time_put : public locale::facet;
```

Parameters

CharType

The type used within a program to encode characters.

OutputIterator

The type of iterator into which the time put functions write their output.

Remarks

As with any locale facet, the static object `ID` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>time_put</code>	The constructor for objects of type <code>time_put</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type that is used to describe a character used by a locale.
<code>iter_type</code>	A type that describes an output iterator.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>do_put</code>	A virtual function that outputs time and date information as a sequence of <code>CharType</code> s.
<code>put</code>	Outputs time and date information as a sequence of <code>CharType</code> s.

Requirements

Header: <locale>

Namespace: std

time_put::char_type

A type that is used to describe a character used by a locale.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

time_put::do_put

A virtual function that outputs time and date information as a sequence of `CharType` s.

```
virtual iter_type do_put(  
    iter_type next,  
    ios_base& _Iosbase,  
    const tm* _Pt,  
    char _Fmt,  
    char _Mod = 0) const;
```

Parameters

next

An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase

Unused.

_Pt

The time and date information being output.

_Fmt

The format of the output. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

_Mod

A modifier for the format. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The virtual protected member function generates sequential elements beginning at `next` from time values stored in the object * `_Pt`, of type `tm`. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by `strftime`, with a last argument of `_Pt`, for generating a series of **char** elements into an array. Each such **char** element is assumed to map to an equivalent element of type `CharType` by a simple, one-to-one mapping. If `_Mod` equals zero, the effective format is "%F", where F is replaced by `_Fmt`. Otherwise, the effective format is "%MF", where M is replaced by `_Mod`.

Example

See the example for [put](#), which calls `do_put`.

time_put::iter_type

A type that describes an output iterator.

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `OutputIterator`.

time_put::put

Outputs time and date information as a sequence of `CharType` s.

```
iter_type put(iter_type next,
              ios_base& _Iosbase,
              char_type _Fill,
              const tm* _Pt,
              char _Fmt,
              char _Mod = 0) const;

iter_type put(iter_type next,
              ios_base& _Iosbase,
              char_type _Fill,
              const tm* _Pt,
              const CharType* first,
              const CharType* last) const;
```

Parameters

next

An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase

Unused.

_Fill

The character of type `CharType` used for spacing.

_Pt

The time and date information being output.

_Fmt

The format of the output. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

_Mod

A modifier for the format. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

first

The beginning of the formatting string for the output. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

last

The end of the formatting string for the output. See [strftime](#), [wcsftime](#), [_strftime_l](#), [_wcsftime_l](#) for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The first member function returns `do_put(next, _Iosbase, _Fill, _Pt, _Fmt, _Mod)`. The second member function copies to * `next` ++ any element in the interval [`first`, `last`) other than a percent (%). For a percent followed by a character C in the interval [`first`, `last`), the function instead evaluates `next = do_put(next, _Iosbase, _Fill, _Pt, C, 0)` and skips past C. If, however, C is a qualifier character from the set EOQ#, followed by a character `c2` in the interval [`first`, `last`), the function instead evaluates `next = do_put(next, _Iosbase, _Fill, _Pt, c2, C)` and skips past `c2`.

Example

```
// time_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
    locale loc;
    basic_stringstream<char> pszPutI;
    ios_base::iostate st = 0;
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    t.tm_hour = 5;
    t.tm_min = 30;
    t.tm_sec = 40;
    t.tm_year = 00;
    t.tm_mday = 4;
    t.tm_mon = 6;

    pszPutI.imbue( loc );
    char *pattern = "x: %X %x";
    use_facet <time_put <char> >
    (loc).put(basic_ostream<char>::_Iter(pszPutI.rdbuf( )),
             pszPutI, ' ', &t, pattern, pattern+strlen(pattern));

    cout << "num_put( ) = " << pszPutI.rdbuf( )->str( ) << endl;

    char strftimebuf[255];
    strftime(&strftimebuf[0], 255, pattern, &t);
    cout << "strftime( ) = " << &strftimebuf[0] << endl;
}
```

```
num_put( ) = x: 05:30:40 07/04/00
strftime( ) = x: 05:30:40 07/04/00
```

time_put::time_put

Constructor for objects of type `time_put`.

```
explicit time_put(size_t _Refs = 0);
```

Parameters

_Refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

The constructor initializes its base object with [locale::facet\(_Refs\)](#).

See also

[<locale>](#)

[time_base Class](#)

[Thread Safety in the C++ Standard Library](#)

time_put_byname Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The derived template class describes an object that can serve as a locale facet of type `time_put< CharType, OutputIterator >`.

Syntax

```
template <class CharType, class OutIt = ostreambuf_iterator<CharType, char_traits<CharType>>>
class time_put_byname : public time_put<CharType, OutputIterator>
{
public:
    explicit time_put_byname(
        const char* _Locname,
        size_t _Refs = 0);

    explicit time_put_byname(
        const string& _Locname,
        size_t _Refs = 0);

protected:
    virtual ~time_put_byname();

};
```

Parameters

_Locname

A locale name.

_Refs

An initial reference count.

Remarks

Its behavior is determined by the [named](#) locale *_Locname*. Each constructor initializes its base object with `time_put<CharType, OutputIterator>(_Refs)`.

Requirements

Header: <locale>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

wbuffer_convert Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes a stream buffer that controls the transmission of elements to and from a byte stream buffer.

Syntax

```
template <class Codecvt, class Elem = wchar_t, class Traits = std::char_traits<Elem>>
class wbuffer_convert
    : public std::basic_streambuf<Elem, Traits>
```

Parameters

PARAMETER	DESCRIPTION
<i>Codecvt</i>	The locale facet that represents the conversion object.
<i>Elem</i>	The wide-character element type.
<i>Traits</i>	The traits associated with <i>Elem</i> .

Remarks

This template class describes a stream buffer that controls the transmission of elements of type `_Elem`, whose character traits are described by the class `Traits`, to and from a byte stream buffer of type `std::streambuf`.

Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

An object of this template class stores:

- A pointer to its underlying byte stream buffer
- A pointer to the allocated conversion object (which is freed when the [wbuffer_convert](#)

wstring_convert Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

The template class `wstring_convert` performs conversions between a wide string and a byte string.

Syntax

```
template <class Codecvt, class Elem = wchar_t>
class wstring_convert
```

Parameters

Codecvt

The [locale](#) facet that represents the conversion object.

Elem

The wide-character element type.

Remarks

The template class describes an object that controls conversions between wide string objects of class `std::basic_string<Elem>` and byte string objects of class `std::basic_string<char>` (also known as `std::string`). The template class defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

An object of this template class stores:

- A byte string to display on errors
- A wide string to display on errors
- A pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- A conversion state object of type [state_type](#)
- A conversion count

Constructors

CONSTRUCTOR	DESCRIPTION
wstring_convert	Constructs an object of type <code>wstring_convert</code> .

Typedefs

TYPE NAME	DESCRIPTION
byte_string	A type that represents a byte string.
wide_string	A type that represents a wide string.

TYPE NAME	DESCRIPTION
<code>state_type</code>	A type that represents the conversion state.
<code>int_type</code>	A type that represents an integer.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>from_bytes</code>	Converts a byte string to a wide string.
<code>to_bytes</code>	Converts a wide string to a byte string.
<code>converted</code>	Returns the number of successful conversions.
<code>state</code>	Returns an object representing the state of the conversion.

Requirements

Header: `<locale>`

Namespace: `std`

`wstring_convert::byte_string`

A type that represents a byte string.

```
typedef std::basic_string<char> byte_string;
```

Remarks

The type is a synonym for `std::basic_string<char>`.

`wstring_convert::converted`

Returns the number of successful conversions.

```
size_t converted() const;
```

Return Value

The number of successful conversions.

Remarks

The number of successful conversions is stored in the conversion count object.

`wstring_convert::from_bytes`

Converts a byte string to a wide string.

```

wide_string from_bytes(char Byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& Bstr);
wide_string from_bytes(const char* first, const char* last);

```

Parameters

PARAMETER	DESCRIPTION
<i>Byte</i>	The single-element byte sequence to be converted.
<i>ptr</i>	The C-style, null-terminated sequence of characters to be converted.
<i>Bstr</i>	The byte_string to be converted.
<i>first</i>	The first character in a range of characters to be converted.
<i>last</i>	The last character in a range of characters to be converted.

Return Value

A wide string object resulting from the conversion.

Remarks

If the [conversion state](#) object was *not* constructed with an explicit value, it is set to its default value (the initial conversion state) before the conversion begins. Otherwise it is left unchanged.

The number of input elements successfully converted is stored in the conversion count object. If no conversion error occurs, the member function returns the converted wide string. Otherwise, if the object was constructed with an initializer for the wide-string error message, the member function returns the wide-string error message object. Otherwise, the member function throws an object of class [range_error](#).

wstring_convert::int_type

A type that represents an integer.

```

typedef typename wide_string::traits_type::int_type int_type;

```

Remarks

The type is a synonym for `wide_string::traits_type::int_type`.

wstring_convert::state

Returns an object representing the state of the conversion.

```

state_type state() const;

```

Return Value

The [conversion state](#) object that represents the state of the conversion.

Remarks

wstring_convert::state_type

A type that represents the conversion state.

```
typedef typename Codecvt::state_type state_type;
```

Remarks

The type describes an object that can represent a conversion state. The type is a synonym for `Codecvt::state_type`.

wstring_convert::to_bytes

Converts a wide string to a byte string.

```
byte_string to_bytes(Elem Char);
byte_string to_bytes(const Elem* Wptr);
byte_string to_bytes(const wide_string& Wstr);
byte_string to_bytes(const Elem* first, const Elem* last);
```

Parameters

PARAMETER	DESCRIPTION
<i>Char</i>	The wide character to be converted.
<i>Wptr</i>	The C-style, null-terminated sequence, beginning at <code>wptr</code> , to be converted.
<i>Wstr</i>	The wide_string to be converted.
<i>first</i>	The first element in a range of elements to be converted.
<i>last</i>	The last element in a range of elements to be converted.

Remarks

If the [conversion state](#) object was *not* constructed with an explicit value, it is set to its default value (the initial conversion state) before the conversion begins. Otherwise it is left unchanged.

The number of input elements successfully converted is stored in the conversion count object. If no conversion error occurs, the member function returns the converted byte string. Otherwise, if the object was constructed with an initializer for the byte-string error message, the member function returns the byte-string error message object. Otherwise, the member function throws an object of class [range_error](#).

wstring_convert::wide_string

A type that represents a wide string.

```
typedef std::basic_string<Elem> wide_string;
```

Remarks

The type is a synonym for `std::basic_string<Elem>`.

wstring_convert::wstring_convert

Constructs an object of type `wstring_convert` .

```
wstring_convert(Codecvt *Pcvt = new Codecvt);
wstring_convert(Codecvt *Pcvt, state_type _State);
wstring_convert(const byte_string& _Berr, const wide_string& Werr = wide_string());
```

Parameters

PARAMETER	DESCRIPTION
<i>*Pcvt</i>	The object of type <code>Codecvt</code> to perform the conversion.
<i>_State</i>	The object of type <code>state_type</code> representing the conversion state.
<i>_Berr</i>	The <code>byte_string</code> to display on errors.
<i>Werr</i>	The <code>wide_string</code> to display on errors.

Remarks

The first constructor stores *Pcvt_arg* in the `conversion object`

<map>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template classes map and multimap and their supporting templates.

Syntax

```
#include <map>
```

Members

Operators

MAP VERSION	MULTIMAP VERSION	DESCRIPTION
<code>operator!= (map)</code>	<code>operator!= (multimap)</code>	Tests if the map or multimap object on the left side of the operator is not equal to the map or multimap object on the right side.
<code>operator< (map)</code>	<code>operator< (multimap)</code>	Tests if the map or multimap object on the left side of the operator is less than the map or multimap object on the right side.
<code>operator<= (map)</code>	<code>operator<= (multimap)</code>	Tests if the map or multimap object on the left side of the operator is less than or equal to the map or multimap object on the right side.
<code>operator== (map)</code>	<code>operator== (multimap)</code>	Tests if the map or multimap object on the left side of the operator is equal to the map or multimap object on the right side.
<code>operator> (map)</code>	<code>operator> (multimap)</code>	Tests if the map or multimap object on the left side of the operator is greater than the map or multimap object on the right side.
<code>operator>= (map)</code>	<code>operator>= (multimap)</code>	Tests if the map or multimap object on the left side of the operator is greater than or equal to the map or multimap object on the right side.

Specialized Template Functions

MAP VERSION	MULTIMAP VERSION	DESCRIPTION
<code>swap (map)</code>	<code>swap (multimap)</code>	Exchanges the elements of two maps or multimaps.

Classes

CLASS	DESCRIPTION
value_compare Class	Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.
map Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a unique key with which the data is automatically ordered.
multimap Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a key with which the data is automatically ordered and the keys do not need to have unique values.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<map> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

[swap \(map\)](#)

[swap \(multimap\)](#)

swap (map)

Exchanges the elements of two maps.

```
template <class key, class T, class _Pr, class _Alloc>
void swap(
    map<Key, Traits, Compare, Allocator>& left,
    map<Key, Traits, Compare, Allocator>& right);
```

Parameters

right

The map providing the elements to be swapped, or the map whose elements are to be exchanged with those of the map *left*.

left

The map whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class `map` to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template < class T> void swap(T&, T&)**, in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function `map::swap` for an example that uses the template version of `swap`.

swap (multimap)

Exchanges the elements of two multimaps.

```
template <class key, class T, class _Pr, class _Alloc>
void swap(
    multimap<Key, Traits, Compare, Allocator>& left,
    multimap<Key, Traits, Compare, Allocator>& right);
```

Parameters

right

The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap *left*.

left

The multimap whose elements are to be exchanged with those of the multimap *right*.

Remarks

The template function is an algorithm specialized on the container class `map` to execute on the container class `multimap` to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template < class T> void swap(T&, T&)**, in the `algorithm` class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function `multimap::swap` for an example that uses the template version of `swap`.

See also

[<map>](#)

<map> operators

3/28/2019 • 15 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>
<code>operator!= (multimap)</code>	<code>operator> (multimap)</code>	<code>operator>= (multimap)</code>
<code>operator< (multimap)</code>	<code>operator<= (multimap)</code>	<code>operator== (multimap)</code>

operator!=

Tests if the map object on the left side of the operator is not equal to the map object on the right side.

```
bool operator!=(  
    const map <Key, Type, Traits, Allocator>& left,  
    const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `map`.

right

An object of type `map`.

Return Value

true if the maps are not equal; **false** if maps are equal.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// map_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1, m2, m3;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 != m2 )
        cout << "The maps m1 and m2 are not equal." << endl;
    else
        cout << "The maps m1 and m2 are equal." << endl;

    if ( m1 != m3 )
        cout << "The maps m1 and m3 are not equal." << endl;
    else
        cout << "The maps m1 and m3 are equal." << endl;
}
/* Output:
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
*/

```

operator<

Tests if the map object on the left side of the operator is less than the map object on the right side.

```

bool operator<(
    const map <Key, Type, Traits, Allocator>& left,
    const map <Key, Type, Traits, Allocator>& right);

```

Parameters

left

An object of type `map`.

right

An object of type `map`.

Return Value

true if the map on the left side of the operator is strictly less than the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// map_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map<int, int> m1, m2, m3;
    int i;
    typedef pair<int, int> Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
    }

    if ( m1 < m2 )
        cout << "The map m1 is less than the map m2." << endl;
    else
        cout << "The map m1 is not less than the map m2." << endl;

    if ( m1 < m3 )
        cout << "The map m1 is less than the map m3." << endl;
    else
        cout << "The map m1 is not less than the map m3." << endl;
}
/* Output:
The map m1 is less than the map m2.
The map m1 is not less than the map m3.
*/

```

operator<=

Tests if the map object on the left side of the operator is less than or equal to the map object on the right side.

```

bool operator<=(
    const map <Key, Type, Traits, Allocator>& left,
    const map <Key, Type, Traits, Allocator>& right);

```

Parameters

left

An object of type `map`.

right

An object of type `map`.

Return Value

true if the map on the left side of the operator is less than or equal to the map on the right side of the operator; otherwise **false**.

Example

```

// map_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1, m2, m3, m4;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
        m4.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 <= m2 )
        cout << "The map m1 is less than or equal to the map m2." << endl;
    else
        cout << "The map m1 is greater than the map m2." << endl;

    if ( m1 <= m3 )
        cout << "The map m1 is less than or equal to the map m3." << endl;
    else
        cout << "The map m1 is greater than the map m3." << endl;

    if ( m1 <= m4 )
        cout << "The map m1 is less than or equal to the map m4." << endl;
    else
        cout << "The map m1 is greater than the map m4." << endl;
}
/* Output:
The map m1 is less than or equal to the map m2.
The map m1 is greater than the map m3.
The map m1 is less than or equal to the map m4.
*/

```

operator==

Tests if the map object on the left side of the operator is equal to the map object on the right side.

```

bool operator==(
    const map <Key, Type, Traits, Allocator>& left,
    const map <Key, Type, Traits, Allocator>& right);

```

Parameters

left

An object of type `map` .

right

An object of type `map` .

Return Value

true if the map on the left side of the operator is equal to the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// map_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map < int, int > m1, m2, m3;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 == m2 )
        cout << "The maps m1 and m2 are equal." << endl;
    else
        cout << "The maps m1 and m2 are not equal." << endl;

    if ( m1 == m3 )
        cout << "The maps m1 and m3 are equal." << endl;
    else
        cout << "The maps m1 and m3 are not equal." << endl;
}
/* Output:
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
*/
```

operator>

Tests if the map object on the left side of the operator is greater than the map object on the right side.

```
bool operator>(
    const map <Key, Type, Traits, Allocator>& left,
    const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `map`.

right

An object of type `map`.

Return Value

true if the map on the left side of the operator is greater than the map on the right side of the operator; otherwise **false**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// map_op_gt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map < int, int > m1, m2, m3;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
    }

    if ( m1 > m2 )
        cout << "The map m1 is greater than the map m2." << endl;
    else
        cout << "The map m1 is not greater than the map m2." << endl;

    if ( m1 > m3 )
        cout << "The map m1 is greater than the map m3." << endl;
    else
        cout << "The map m1 is not greater than the map m3." << endl;
}
/* Output:
The map m1 is not greater than the map m2.
The map m1 is greater than the map m3.
*/
```

operator>=

Tests if the map object on the left side of the operator is greater than or equal to the map object on the right side.

```
bool operator>=(
    const map <Key, Type, Traits, Allocator>& left,
    const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `map` .

right

An object of type `map` .

Return Value

true if the map on the left side of the operator is greater than or equal to the map on the right side of the list; otherwise **false**.

Example

```

// map_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map < int, int > m1, m2, m3, m4;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
        m4.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 >= m2 )
        cout << "Map m1 is greater than or equal to map m2." << endl;
    else
        cout << "The map m1 is less than the map m2." << endl;

    if ( m1 >= m3 )
        cout << "Map m1 is greater than or equal to map m3." << endl;
    else
        cout << "The map m1 is less than the map m3." << endl;

    if ( m1 >= m4 )
        cout << "Map m1 is greater than or equal to map m4." << endl;
    else
        cout << "The map m1 is less than the map m4." << endl;
}
/* Output:
The map m1 is less than the map m2.
Map m1 is greater than or equal to map m3.
Map m1 is greater than or equal to map m4.
*/

```

operator!= (multimap)

Tests if the multimap object on the left side of the operator is not equal to the multimap object on the right side.

```

bool operator!=(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);

```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimaps are not equal; **false** if multimaps are equal.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps

are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multimap_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1, m2, m3;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 != m2 )
        cout << "The multimaps m1 and m2 are not equal." << endl;
    else
        cout << "The multimaps m1 and m2 are equal." << endl;

    if ( m1 != m3 )
        cout << "The multimaps m1 and m3 are not equal." << endl;
    else
        cout << "The multimaps m1 and m3 are equal." << endl;
}
/* Output:
The multimaps m1 and m2 are not equal.
The multimaps m1 and m3 are equal.
*/
```

operator<

Tests if the multimap object on the left side of the operator is less than the multimap object on the right side.

```
bool operator<(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimap on the left side of the operator is strictly less than the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The less-than

relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap < int, int > m1, m2, m3;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
    }

    if ( m1 < m2 )
        cout << "The multimap m1 is less than the multimap m2." << endl;
    else
        cout << "The multimap m1 is not less than the multimap m2." << endl;

    if ( m1 < m3 )
        cout << "The multimap m1 is less than the multimap m3." << endl;
    else
        cout << "The multimap m1 is not less than the multimap m3." << endl;
}
/* Output:
The multimap m1 is less than the multimap m2.
The multimap m1 is not less than the multimap m3.
*/
```

operator<=

Tests if the multimap object on the left side of the operator is less than or equal to the multimap object on the right side.

```
bool operator<=(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimap on the left side of the operator is less than or equal to the multimap on the right side of the operator; otherwise **false**.

Example

```

// multimap_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1, m2, m3, m4;
    int i;
    typedef pair <int, int> Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
        m4.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 <= m2 )
        cout << "m1 is less than or equal to m2" << endl;
    else
        cout << "m1 is greater than m2" << endl;

    if ( m1 <= m3 )
        cout << "m1 is less than or equal to m3" << endl;
    else
        cout << "m1 is greater than m3" << endl;

    if ( m1 <= m4 )
        cout << "m1 is less than or equal to m4" << endl;
    else
        cout << "m1 is greater than m4" << endl;
}
/* Output:
m1 is less than or equal to m2
m1 is greater than m3
m1 is less than or equal to m4
*/

```

operator==

Tests if the multimap object on the left side of the operator is equal to the multimap object on the right side.

```

bool operator==(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);

```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimap on the left side of the operator is equal to the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multimap_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> m1, m2, m3;
    int i;
    typedef pair<int, int> Int_Pair;

    for (i = 0; i < 3; i++)
    {
        m1.insert(Int_Pair(i, i));
        m2.insert(Int_Pair(i, i*i));
        m3.insert(Int_Pair(i, i));
    }

    if ( m1 == m2 )
        cout << "m1 and m2 are equal" << endl;
    else
        cout << "m1 and m2 are not equal" << endl;

    if ( m1 == m3 )
        cout << "m1 and m3 are equal" << endl;
    else
        cout << "m1 and m3 are not equal" << endl;
}
/* Output:
m1 and m2 are not equal
m1 and m3 are equal
*/
```

operator>

Tests if the multimap object on the left side of the operator is greater than the multimap object on the right side.

```
bool operator>(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimap on the left side of the operator is greater than the multimap on the right side of the operator; otherwise **false**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multimap_op_gt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap < int, int > m1, m2, m3;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 0 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
    }

    if ( m1 > m2 )
        cout << "The multimap m1 is greater than the multimap m2." << endl;
    else
        cout << "Multimap m1 is not greater than multimap m2." << endl;

    if ( m1 > m3 )
        cout << "The multimap m1 is greater than the multimap m3." << endl;
    else
        cout << "The multimap m1 is not greater than the multimap m3." << endl;
}
/* Output:
Multimap m1 is not greater than multimap m2.
The multimap m1 is greater than the multimap m3.
*/
```

operator>=

Tests if the multimap object on the left side of the operator is greater than or equal to the multimap object on the right side.

```
bool operator>=(
    const multimap <Key, Type, Traits, Allocator>& left,
    const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `multimap`.

right

An object of type `multimap`.

Return Value

true if the multimap on the left side of the operator is greater than or equal to the multimap on the right side of the list; otherwise **false**.

Example

```

// multimap_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap < int, int > m1, m2, m3, m4;
    int i;
    typedef pair < int, int > Int_Pair;

    for ( i = 1 ; i < 3 ; i++ )
    {
        m1.insert ( Int_Pair ( i, i ) );
        m2.insert ( Int_Pair ( i, i * i ) );
        m3.insert ( Int_Pair ( i, i - 1 ) );
        m4.insert ( Int_Pair ( i, i ) );
    }

    if ( m1 >= m2 )
        cout << "The multimap m1 is greater than or equal to the multimap m2." << endl;
    else
        cout << "The multimap m1 is less than the multimap m2." << endl;

    if ( m1 >= m3 )
        cout << "The multimap m1 is greater than or equal to the multimap m3." << endl;
    else
        cout << "The multimap m1 is less than the multimap m3." << endl;

    if ( m1 >= m4 )
        cout << "The multimap m1 is greater than or equal to the multimap m4." << endl;
    else
        cout << "The multimap m1 is less than the multimap m4." << endl;
}
/* Output:
The multimap m1 is less than the multimap m2.
The multimap m1 is greater than or equal to the multimap m3.
The multimap m1 is greater than or equal to the multimap m4.
*/

```

See also

[<map>](#)

map Class

11/15/2018 • 59 minutes to read • [Edit Online](#)

Used for the storage and retrieval of data from a collection in which each element is a pair that has both a data value and a sort key. The value of the key is unique and is used to automatically sort the data.

The value of an element in a map can be changed directly. The key value is a constant and cannot be changed. Instead, key values associated with old elements must be deleted, and new key values must be inserted for new elements.

Syntax

```
template <class Key,  
          class Type,  
          class Traits = less<Key>,  
          class Allocator=allocator<pair <const Key, Type>>>  
class map;
```

Parameters

Key

The key data type to be stored in the map.

Type

The element data type to be stored in the map.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the map. This argument is optional and the binary predicate `less<Key>` is the default value.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Allocator

The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is `allocator<pair<const Key, Type>>`.

Remarks

The C++ Standard Library map class is:

- A container of variable size that efficiently retrieves element values based on associated key values.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values according to a specified comparison function.
- Unique. because each of its elements must have a unique key.
- A pair-associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and independent of element or key type. The data types used for elements and keys are specified as parameters in the class template together with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the [insert](#) and [map](#) class member functions have versions that take as template parameters a weaker input iterator, whose functionality requirements are fewer than those guaranteed by the class of bidirectional iterators. The different iterator concepts are related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with it must be limited by those requirements. An input iterator may be dereferenced to refer to some object and may be incremented to the next iterator in the sequence.

We recommend that you base the choice of container type on the kind of searching and inserting that is required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations perform them in a worst-case time that is proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that specifically pointed to the removed elements.

We recommend that you make the map the associative container of choice when conditions that associate values with keys are satisfied by the application. A model for this kind of structure is an ordered list of uniquely occurring key words that have associated string values that provide definitions. If a word has more than one correct definition, so that key is not unique, then a multimap would be the container of choice. If just the list of words is being stored, then a set would be the appropriate container. If multiple occurrences of the words are allowed, then a multiset would be appropriate.

The map orders the elements it controls by calling a stored function object of type [key_compare](#). This stored object is a comparison function that is accessed by calling the [key_comp](#) method. In general, any two given elements are compared to determine whether one is less than the other or whether they are equivalent. As all elements are compared, an ordered sequence of non-equivalent elements is created.

NOTE

The comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x,y)$ is a function object that has two argument objects x and y , and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive, and if equivalence is transitive, where two objects x and y are defined to be equivalent when both $f(x,y)$ and $f(y,x)$ are **false**. If the stronger condition of equality between keys replaces that of equivalence, the ordering becomes total (in the sense that all the elements are ordered with regard to one other), and the keys matched will be indiscernible from one other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Members

Constructors

CONSTRUCTOR	DESCRIPTION
map	Constructs a list of a specific size or with elements of a specific value or with a specific <code>allocator</code> or as a copy of some other map.

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	A typedef for the <code>allocator</code> class for the map object.
const_iterator	A typedef for a bidirectional iterator that can read a const element in the map.

TYPE NAME	DESCRIPTION
<code>const_pointer</code>	A typedef for a pointer to a const element in a map.
<code>const_reference</code>	A typedef for a reference to a const element stored in a map for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in the map.
<code>difference_type</code>	A signed integer typedef for the number of elements of a map in a range between elements pointed to by iterators.
<code>iterator</code>	A typedef for a bidirectional iterator that can read or modify any element in a map.
<code>key_compare</code>	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the map.
<code>key_type</code>	A typedef for the sort key stored in each element of the map.
<code>mapped_type</code>	A typedef for the data stored in each element of a map.
<code>pointer</code>	A typedef for a pointer to a const element in a map.
<code>reference</code>	A typedef for a reference to an element stored in a map.
<code>reverse_iterator</code>	A typedef for a bidirectional iterator that can read or modify an element in a reversed map.
<code>size_type</code>	An unsigned integer typedef for the number of elements in a map
<code>value_type</code>	A typedef for the type of object stored as an element in a map.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>at</code>	Finds an element with a specified key value.
<code>begin</code>	Returns an iterator that points to the first element in the map.
<code>cbegin</code>	Returns a const iterator that points to the first element in the map.
<code>cend</code>	Returns a const past-the-end iterator.
<code>clear</code>	Erases all the elements of a map.
<code>count</code>	Returns the number of elements in a map whose key matches the key specified in a parameter.

MEMBER FUNCTION	DESCRIPTION
<code>crbegin</code>	Returns a const iterator that points to the first element in a reversed map.
<code>crend</code>	Returns a const iterator that points to the location after the last element in a reversed map.
<code>emplace</code>	Inserts an element constructed in place into the map.
<code>emplace_hint</code>	Inserts an element constructed in place into the map, with a placement hint.
<code>empty</code>	Returns true if a map is empty.
<code>end</code>	Returns the past-the-end iterator.
<code>equal_range</code>	Returns a pair of iterators. The first iterator in the pair points to the first element in a <code>map</code> with a key that is greater than a specified key. The second iterator in the pair points to the first element in the <code>map</code> with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a map from the specified positions.
<code>find</code>	Returns an iterator that points to the location of an element in a map that has a key equal to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object that is used to construct the map.
<code>insert</code>	Inserts an element or a range of elements into the map at a specified position.
<code>key_comp</code>	Returns a copy of the comparison object that used to order keys in a map.
<code>lower_bound</code>	Returns an iterator to the first element in a map that has a key value that is equal to or greater than that of a specified key.
<code>max_size</code>	Returns the maximum length of the map.
<code>rbegin</code>	Returns an iterator that points to the first element in a reversed map.
<code>rend</code>	Returns an iterator that points to the location after the last element in a reversed map.
<code>size</code>	Returns the number of elements in the map.
<code>swap</code>	Exchanges the elements of two maps.

MEMBER FUNCTION	DESCRIPTION
upper_bound	Returns an iterator to the first element in a map that has a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object that is used to order element values in a map.

Operators

OPERATOR	DESCRIPTION
operator[]	Inserts an element into a map with a specified key value.
operator=	Replaces the elements of a map with a copy of another map.

Requirements

Header: <map>

Namespace: std

map::allocator_type

A type that represents the allocator class for the map object.

```
typedef Allocator allocator_type;
```

Example

See example for [get_allocator](#) for an example that uses `allocator_type`.

map::at

Finds an element with a specified key value.

```
Type& at(const Key& key);

const Type& at(const Key& key) const;
```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>key</i>	The key value to find.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value is not found, then the function throws an object of class [out_of_range Class](#).

Example

```
// map_at.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

typedef std::map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // find and show elements
    std::cout << "c1.at('a') == " << c1.at('a') << std::endl;
    std::cout << "c1.at('b') == " << c1.at('b') << std::endl;
    std::cout << "c1.at('c') == " << c1.at('c') << std::endl;

    return (0);
}
```

map::begin

Returns an iterator addressing the first element in the map.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the map or the location succeeding an empty map.

Example

```

// map_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;

    map <int, int> :: iterator m1_Iter;
    map <int, int> :: const_iterator m1_cIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 0, 0 ) );
    m1.insert ( Int_Pair ( 1, 1 ) );
    m1.insert ( Int_Pair ( 2, 4 ) );

    m1_cIter = m1.begin ( );
    cout << "The first element of m1 is " << m1_cIter -> first << endl;

    m1_Iter = m1.begin ( );
    m1.erase ( m1_Iter );

    // The following 2 lines would err because the iterator is const
    // m1_cIter = m1.begin ( );
    // m1.erase ( m1_cIter );

    m1_cIter = m1.begin( );
    cout << "The first element of m1 is now " << m1_cIter -> first << endl;
}

```

```

The first element of m1 is 0
The first element of m1 is now 1

```

map::cbegin

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cbegin() const;
```

Return Value

A **const** bidirectional iterator addressing the first element in the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the [auto](#) type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

map::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

map::clear

Erases all the elements of a map.

```
void clear();
```

Example

The following example demonstrates the use of the `map::clear` member function.

```
// map_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 4));

    i = m1.size();
    cout << "The size of the map is initially "
         << i << "." << endl;

    m1.clear();
    i = m1.size();
    cout << "The size of the map after clearing is "
         << i << "." << endl;
}
```

```
The size of the map is initially 2.
The size of the map after clearing is 0.
```

map::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the map.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by map points to elements that are objects of [value_type](#), that is of type `pair < constKey, Type >`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_iterator cIter` pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `cIter -> first`, which is equivalent to `(* cIter).first`.

To access the value of the mapped datum for the element, use `cIter -> second`, which is equivalent to `(* cIter).second`.

Example

See example for [begin](#) for an example that uses `const_iterator`.

map::const_pointer

A type that provides a pointer to a **const** element in a map.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a map object.

map::const_reference

A type that provides a reference to a **const** element stored in a map for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// map_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( m1.begin( ) -> first );

    // The following line would cause an error as the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the map is "
         << Ref2 << "." << endl;
}
```

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
```

map::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the map.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the map in reverse.

The `const_reverse_iterator` defined by map points to elements that are objects of `value_type`, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator crIter` pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `crIter -> first`, which is equivalent to `(*crIter).first`.

To access the value of the mapped datum for the element, use `crIter -> second`, which is equivalent to `(*crIter).second`.

Example

See the example for [rend](#) for an example of how to declare and use `const_reverse_iterator`.

map::count

Returns the number of elements in a map whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key value of the elements to be matched from the map.

Return Value

1 if the map contains an element whose sort key matches the parameter key; 0 if the map does not contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

`[lower_bound(key), upper_bound(key))`

which is 0 or 1 in the case of map, which is a unique associative container.

Example

The following example demonstrates the use of the `map::count` member function.

```

// map_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Keys must be unique in map, so duplicates are ignored
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}

```

```

The number of elements in m1 with a sort key of 1 is: 1.
The number of elements in m1 with a sort key of 2 is: 1.
The number of elements in m1 with a sort key of 3 is: 0.

```

map::crbegin

Returns a const iterator addressing the first element in a reversed map.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [map](#) or addressing what had been the last element in the unreversed [map](#).

Remarks

[crbegin](#) is used with a reversed [map](#) just as [begin](#) is used with a [map](#).

With the return value of [crbegin](#), the [map](#) object cannot be modified

[crbegin](#) can be used to iterate through a [map](#) backwards.

Example

```
// map_crbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;

    map <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_crIter = m1.crbegin( );
    cout << "The first element of the reversed map m1 is "
         << m1_crIter -> first << "." << endl;
}
```

The first element of the reversed map m1 is 3.

map::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed map.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [map](#) (the location that had preceded the first element in the unreversed [map](#)).

Remarks

[crend](#) is used with a reversed map just as [end](#) is used with a [map](#) .

With the return value of [crend](#) , the [map](#) object cannot be modified.

[crend](#) can be used to test to whether a reverse iterator has reached the end of its [map](#) .

The value returned by [crend](#) should not be dereferenced.

Example

```

// map_crend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;

    map <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_crIter = m1.crend( );
    m1_crIter--;
    cout << "The last element of the reversed map m1 is "
         << m1_crIter -> first << "." << endl;
}

```

The last element of the reversed map m1 is 1.

map::difference_type

A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.

```
typedef allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range $[first, last)$ between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random access iterators provided by a random access container such as vector.

Example

```

// map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
    using namespace std;
    map <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 3, 20 ) );
    m1.insert ( Int_Pair ( 2, 30 ) );

    map <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
    m1_bIter = m1.begin( );
    m1_eIter = m1.end( );

    // Count the number of elements in a map
    map <int, int>::difference_type df_count = 1;
    m1_Iter = m1.begin( );
    while ( m1_Iter != m1_eIter)
    {
        df_count++;
        m1_Iter++;
    }

    cout << "The number of elements in the map m1 is: "
         << df_count << "." << endl;
}

```

The number of elements in the map m1 is: 4.

map::emplace

Inserts an element constructed in place (no copy or move operations are performed) into a map.

```

template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>args</i>	The arguments forwarded to construct an element to be inserted into the map unless it already contains an element whose value is equivalently ordered.

Return Value

A [pair](#) whose **bool** component is true if an insertion was made, and false if the map already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted

element if the **bool** component is true, or to the existing element if the **bool** component is false.

To access the iterator component of a `pair` `pr`, use `pr.first`; to dereference it, use `*pr.first`. To access the **bool** component, use `pr.second`. For an example, see the sample code later in this article.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

```

// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    map<int, string> m1;

    auto ret = m1.emplace(10, "ten");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
              << endl << " The existing element is (" << pr.first << ", " << pr.second << ")"
              << endl;
        cout << "map not modified" << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;

    ret = m1.emplace(10, "one zero");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
              << endl << " The existing element is (" << pr.first << ", " << pr.second << ")"
              << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;
}

```

map::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```

template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>args</i>	The arguments forwarded to construct an element to be inserted into the map unless the map already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.
<i>where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>where</i> , insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element with its key.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example


```

// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: " << endl;

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    map<string, string> m1;

    // Emplace some test data
    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "map starting data: ";
    print(m1);
    cout << endl;

    // Emplace with hint
    // m1.end() should be the "next" element after this emplacement
    m1.emplace_hint(m1.end(), "Doug", "Engineering");

    cout << "map modified, now contains ";
    print(m1);
    cout << endl;
}

```

map::empty

Tests if a map is empty.

```
bool empty() const;
```

Return Value

true if the map is empty; **false** if the map is nonempty.

Example

```
// map_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1, m2;

    typedef pair <int, int> Int_Pair;
    m1.insert ( Int_Pair ( 1, 1 ) );

    if ( m1.empty( ) )
        cout << "The map m1 is empty." << endl;
    else
        cout << "The map m1 is not empty." << endl;

    if ( m2.empty( ) )
        cout << "The map m2 is empty." << endl;
    else
        cout << "The map m2 is not empty." << endl;
}
```

```
The map m1 is not empty.
The map m2 is empty.
```

map::end

Returns the past-the-end iterator.

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the map is empty, then `map::end() == map::begin()`.

Remarks

`end` is used to test whether an iterator has passed the end of its map.

The value returned by `end` should not be dereferenced.

For a code example, see [map::find](#).

map::equal_range

Returns a pair of iterators that represent the [lower_bound](#) of the key and the [upper_bound](#) of the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

To access the first iterator of a pair `pr` returned by the member function, use `pr.first`, and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second`, and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```
// map_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    typedef map <int, int, less<int> > IntMap;
    IntMap m1;
    map <int, int> :: const_iterator m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
    p1 = m1.equal_range( 2 );

    cout << "The lower bound of the element with "
         << "a key of 2 in the map m1 is: "
         << p1.first -> second << "." << endl;

    cout << "The upper bound of the element with "
         << "a key of 2 in the map m1 is: "
         << p1.second -> second << "." << endl;

    // Compare the upper_bound called directly
    m1_RcIter = m1.upper_bound( 2 );

    cout << "A direct call of upper_bound( 2 ) gives "
         << m1_RcIter -> second << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 2 )." << endl;

    p2 = m1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
        cout << "The map m1 doesn't have an element "
             << "with a key less than 40." << endl;
    else
        cout << "The element of map m1 with a key >= 40 is: "
             << p2.first -> first << "." << endl;
}
```

The lower bound of the element with a key of 2 in the map m1 is: 20.
The upper bound of the element with a key of 2 in the map m1 is: 30.
A direct call of upper_bound(2) gives 30,
matching the 2nd element of the pair returned by equal_range(2).
The map m1 doesn't have an element with a key less than 40.

map::erase

Removes an element or a range of elements in a map from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the map.

Example

```
// map_erase.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions
#include <utility>   // make_pair()

using namespace std;

using mymap = map<int, string>;

void printmap(const mymap& m) {
    for (const auto& elem : m) {
        cout << " [" << elem.first << ", " << elem.second << "]\n";
    }
    cout << endl << "size() == " << m.size() << endl << endl;
}

int main()
{
    mymap m1;

    // Fill in some data to test with, one at a time
    m1.insert(make_pair(1, "A"));
    m1.insert(make_pair(2, "B"));
    m1.insert(make_pair(3, "C"));
    m1.insert(make_pair(4, "D"));
    m1.insert(make_pair(5, "E"));
```

```

cout << "Starting data of map m1 is:" << endl;
printmap(m1);
// The 1st member function removes an element at a given position
m1.erase(next(m1.begin()));
cout << "After the 2nd element is deleted, the map m1 is:" << endl;
printmap(m1);

// Fill in some data to test with, one at a time, using an initializer list
mymap m2
{
    { 10, "Bob" },
    { 11, "Rob" },
    { 12, "Robert" },
    { 13, "Bert" },
    { 14, "Bobby" }
};

cout << "Starting data of map m2 is:" << endl;
printmap(m2);
// The 2nd member function removes elements
// in the range [First, Last)
m2.erase(next(m2.begin()), prev(m2.end()));
cout << "After the middle elements are deleted, the map m2 is:" << endl;
printmap(m2);

mymap m3;

// Fill in some data to test with, one at a time, using emplace
m3.emplace(1, "red");
m3.emplace(2, "yellow");
m3.emplace(3, "blue");
m3.emplace(4, "green");
m3.emplace(5, "orange");
m3.emplace(6, "purple");
m3.emplace(7, "pink");

cout << "Starting data of map m3 is:" << endl;
printmap(m3);
// The 3rd member function removes elements with a given Key
mymap::size_type count = m3.erase(2);
// The 3rd member function also returns the number of elements removed
cout << "The number of elements removed from m3 is: " << count << "." << endl;
cout << "After the element with a key of 2 is deleted, the map m3 is:" << endl;
printmap(m3);
}

```

map::find

Returns an iterator that refers to the location of an element in a map that has a key equivalent to a specified key.

```

iterator find(const Key& key);

const_iterator find(const Key& key) const;

```

Parameters

key

The key value to be matched by the sort key of an element from the map being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the map (`map::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the map whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the map object cannot be modified. If the return value of `find` is assigned to an `iterator`, the map object can be modified

Example

```

// compile with: /EHsc /W4 /MTd
#include <map>
#include <iostream>
#include <vector>
#include <string>
#include <utility> // make_pair()

using namespace std;

template <typename A, typename B> void print_elem(const pair<A, B>& p) {
    cout << "(" << p.first << ", " << p.second << ")" << endl;
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    map<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting map m1 is (key, value):" << endl;
    print_collection(m1);

    vector<pair<int, string>> v;
    v.push_back(make_pair(43, "Tc"));
    v.push_back(make_pair(41, "Nb"));
    v.push_back(make_pair(46, "Pd"));
    v.push_back(make_pair(42, "Mo"));
    v.push_back(make_pair(44, "Ru"));
    v.push_back(make_pair(44, "Ru")); // attempt a duplicate

    cout << "Inserting the following vector data into m1:" << endl;
    print_collection(v);

    m1.insert(v.begin(), v.end());

    cout << "The modified map m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
    findit(m1, 6);
}

```

map::get_allocator

Returns a copy of the allocator object used to construct the map.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the map.

Remarks

Allocators for the map class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// map_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int>::allocator_type m1_Alloc;
    map <int, int>::allocator_type m2_Alloc;
    map <int, double>::allocator_type m3_Alloc;
    map <int, int>::allocator_type m4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    map <int, int> m1;
    map <int, int, allocator<int> > m2;
    map <int, double, allocator<double> > m3;

    m1_Alloc = m1.get_allocator( );
    m2_Alloc = m2.get_allocator( );
    m3_Alloc = m3.get_allocator( );

    cout << "The number of integers that can be allocated\n"
         << "before free memory is exhausted: "
         << m2.max_size( ) << ".\n" << endl;

    cout << "The number of doubles that can be allocated\n"
         << "before free memory is exhausted: "
         << m3.max_size( ) << ".\n" << endl;

    // The following line creates a map m4
    // with the allocator of map m1.
    map <int, int> m4( less<int>( ), m1_Alloc );

    m4_Alloc = m4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated with the other
    if( m1_Alloc == m4_Alloc )
    {
        cout << "The allocators are interchangeable." << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable." << endl;
    }
}
```

map::insert

Inserts an element or a range of elements into a map.


```

// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);

```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>Val</i>	The value of an element to be inserted into the map unless it already contains an element whose key is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>Where</i> , insertion can occur in amortized constant time instead of logarithmic time.)
<i>ValTy</i>	Template parameter that specifies the argument type that the map can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a [pair](#) whose **bool** component is true if an insertion was made, and false if the map already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is true, or to the existing element if the **bool** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the map or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the **bool** component, use `pr.second`. For an example, see the sample code later in this article.

The [value_type](#) of a container is a typedef that belongs to the container, and for map, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into a map that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `m`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [map::emplace](#) and [map::emplace_hint](#).

Example

```
// map_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>
#include <vector>
#include <utility> // make_pair()

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
```

```

// insert single values
map<int, int> m1;
// call insert(const value_type&) version
m1.insert({ 1, 10 });
// call insert(ValTy&&) version
m1.insert(make_pair(2, 20));

cout << "The original key and mapped values of m1 are:" << endl;
print(m1);

// intentionally attempt a duplicate, single element
auto ret = m1.insert(make_pair(1, 111));
if (!ret.second){
    auto pr = *ret.first;
    cout << "Insert failed, element with key value 1 already exists."
        << endl << " The existing element is (" << pr.first << ", " << pr.second << ")"
        << endl;
}
else{
    cout << "The modified key and mapped values of m1 are:" << endl;
    print(m1);
}
cout << endl;

// single element, with hint
m1.insert(m1.end(), make_pair(3, 30));
cout << "The modified key and mapped values of m1 are:" << endl;
print(m1);
cout << endl;

// The templated version inserting a jumbled range
map<int, int> m2;
vector<pair<int, int>> v;
v.push_back(make_pair(43, 294));
v.push_back(make_pair(41, 262));
v.push_back(make_pair(45, 330));
v.push_back(make_pair(42, 277));
v.push_back(make_pair(44, 311));

cout << "Inserting the following vector data into m2:" << endl;
print(v);

m2.insert(v.begin(), v.end());

cout << "The modified key and mapped values of m2 are:" << endl;
print(m2);
cout << endl;

// The templated versions move-constructing elements
map<int, string> m3;
pair<int, string> ip1(475, "blue"), ip2(510, "green");

// single element
m3.insert(move(ip1));
cout << "After the first move insertion, m3 contains:" << endl;
print(m3);

// single element with hint
m3.insert(m3.end(), move(ip2));
cout << "After the second move insertion, m3 contains:" << endl;
print(m3);
cout << endl;

map<int, int> m4;
// Insert the elements from an initializer_list
m4.insert({ { 4, 44 }, { 2, 22 }, { 3, 33 }, { 1, 11 }, { 5, 55 } });
cout << "After initializer_list insertion, m4 contains:" << endl;
print(m4);
cout << endl;

```

```
}
```

map::iterator

A type that provides a bidirectional iterator that can read or modify any element in a map.

```
typedef implementation-defined iterator;
```

Remarks

The iterator defined by map points to elements that are objects of `value_type`, that is of type

`pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an iterator *Iter* pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).second`.

Example

See example for [begin](#) for an example of how to declare and use `iterator`.

map::key_comp

Retrieves a copy of the comparison object used to order keys in a map.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a map uses to order its elements.

Remarks

The stored object defines the member function

```
bool operator(const Key& left, const Key& right);
```

which returns **true** if `left` precedes and is not equal to `right` in the sort order.

Example

```

// map_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;

    map <int, int, less<int> > m1;
    map <int, int, less<int> >::key_compare kc1 = m1.key_comp( );
    bool result1 = kc1( 2, 3 );
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
              << "where kc1 is the function object of m1."
              << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
              << "where kc1 is the function object of m1."
              << endl;
    }

    map <int, int, greater<int> > m2;
    map <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
    bool result2 = kc2( 2, 3 );
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true, "
              << "where kc2 is the function object of m2."
              << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
              << "where kc2 is the function object of m2."
              << endl;
    }
}

```

```

kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.

```

map::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the [map Class](#) topic.

Example

See example for [key_comp](#) for an example of how to declare and use `key_compare`.

map::key_type

A type that describes the sort key stored in each element of the map.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

For more information on *Key*, see the Remarks section of the [map Class](#) topic.

Example

See example for [value_type](#) for an example of how to declare and use `key_type`.

map::lower_bound

Returns an iterator to the first element in a map with a key value that is equal to or greater than that of a specified key.

```
iterator lower_bound(const Key& key);  
  
const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the map object cannot be modified. If the return value of `lower_bound` is assigned to an `iterator`, the map object can be modified.

Example

```

// map_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;
    map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_RcIter = m1.lower_bound( 2 );
    cout << "The first element of map m1 with a key of 2 is: "
         << m1_RcIter -> second << "." << endl;

    // If no match is found for this key, end( ) is returned
    m1_RcIter = m1.lower_bound ( 4 );

    if ( m1_RcIter == m1.end( ) )
        cout << "The map m1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of map m1 with a key of 4 is: "
             << m1_RcIter -> second << "." << endl;

    // The element at a specific location in the map can be found
    // using a dereferenced iterator addressing the location
    m1_AcIter = m1.end( );
    m1_AcIter--;
    m1_RcIter = m1.lower_bound ( m1_AcIter -> first );
    cout << "The element of m1 with a key matching "
         << "that of the last element is: "
         << m1_RcIter -> second << "." << endl;
}

```

```

The first element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.

```

map::map

Constructs a map that is empty or that is a copy of all or part of some other map.

```

map();

explicit map(
    const Traits& Comp);

map(
    const Traits& Comp,
    const Allocator& Al);

map(
    const map& Right);

map(
    map&& Right);

map(
    initializer_list<value_type> IList);

map(
    initializer_list<value_type> IList,
    const Traits& Comp);

map(
    initializer_list<value_type> IList,
    const Traits& Comp,
    const Allocator& Allocator);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>Al</i>	The storage allocator class to be used for this map object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the map, which defaults to <code>hash_compare</code> .
<i>Right</i>	The map of which the constructed set is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>Last</i>	The position of the first element beyond the range of elements to be copied.
<i>lList</i>	The initializer_list from which the elements are to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the map and that can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their map.

All constructors store a function object of type Traits that is used to establish an order among the keys of the map and that can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial map, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the map *Right*.

The fifth constructor specifies a copy of the map by moving *Right*.

The sixth, seventh, and eighth constructors use an initializer_list from which to copy the members.

The next three constructors copy the range [First, Last) of a map with increasing explicitness in specifying the type of comparison function of class Traits and allocator.

Example

```
// map_map.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    typedef pair <int, int> Int_Pair;
    map <int, int>::iterator m1_Iter, m3_Iter, m4_Iter, m5_Iter, m6_Iter, m7_Iter;
    map <int, int, less<int> >::iterator m2_Iter;

    // Create an empty map m0 of key type integer
    map <int, int> m0;

    // Create an empty map m1 with the key comparison
    // function of less than, then insert 4 elements
    map <int, int, less<int> > m1;
    m1.insert(Int_Pair(1, 10));
    m1.insert(Int_Pair(2, 20));
    m1.insert(Int_Pair(3, 30));
    m1.insert(Int_Pair(4, 40));

    // Create an empty map m2 with the key comparison
    // function of greater than, then insert 2 elements
    map <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a map m3 with the
    // allocator of map m1
```

```

map<int, int>::allocator_type m1_Alloc;
m1_Alloc = m1.get_allocator();
map<int, int> m3(less<int>(), m1_Alloc);
m3.insert(Int_Pair(3, 30));

// Create a copy, map m4, of map m1
map<int, int> m4(m1);

// Create a map m5 by copying the range m1[ first, last)
map<int, int>::const_iterator m1_bcIter, m1_ecIter;
m1_bcIter = m1.begin();
m1_ecIter = m1.begin();
m1_ecIter++;
m1_ecIter++;
map<int, int> m5(m1_bcIter, m1_ecIter);

// Create a map m6 by copying the range m4[ first, last)
// and with the allocator of map m2
map<int, int>::allocator_type m2_Alloc;
m2_Alloc = m2.get_allocator();
map<int, int> m6(m4.begin(), ++m4.begin(), less<int>(), m2_Alloc);

cout << "m1 =";
for (auto i : m1)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m2 =";
for(auto i : m2)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m3 =";
for (auto i : m3)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m4 =";
for (auto i : m4)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m5 =";
for (auto i : m5)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m6 =";
for (auto i : m6)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a map m7 by moving m5
cout << "m7 =";
map<int, int> m7(move(m5));
for (auto i : m7)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a map m8 by copying in an initializer_list
map<int, int> m8{ { { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 } } };
cout << "m8: = ";
for (auto i : m8)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a map m9 with an initializer_list and a comparator
map<int, int> m9({ { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 } }, less<int>());
cout << "m9: = ";

```

```

    cout << m9;
    for (auto i : m9)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m10 with an initializer_list, a comparator, and an allocator
    map<int, int> m10({ { 9, 9 }, { 10, 10 }, { 11, 11 }, { 12, 12 } }, less<int>(), m9.get_allocator());
    cout << "m10: = ";
    for (auto i : m10)
        cout << i.first << " " << i.second << ", ";
    cout << endl;
}

```

map::mapped_type

A type that represents the data stored in a map.

```
typedef Type mapped_type;
```

Remarks

The type `mapped_type` is a synonym for the class's *Type* template parameter.

For more information on *Type* see the [map Class](#) topic.

Example

See example for [value_type](#) for an example of how to declare and use `mapped_type`.

map::max_size

Returns the maximum length of the map.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the map.

Example

```

// map_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;
    map <int, int> :: size_type i;

    i = m1.max_size( );
    cout << "The maximum possible length "
        << "of the map is " << i << "."
        << endl << "(Magnitude is machine specific.);";
}

```

map::operator[]

Inserts an element into a map with a specified key value.

```
Type& operator[](const Key& key);
```

```
Type& operator[](Key&& key);
```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>key</i>	The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map `m` using `m[key] = DataValue;` where `DataValue` is the value of the `mapped_type` of the element with a key value of `key`.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions [find](#) and [insert](#) can be used to determine whether an element with a specified key is already present before an insertion.

Example

```

// map_op_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>

int main( )
{
    using namespace std;
    typedef pair <const int, int> cInt2Int;
    map <int, int> m1;
    map <int, int> :: iterator pIter;

    // Insert a data value of 10 with a key of 1
    // into a map using the operator[] member function
    m1[ 1 ] = 10;

    // Compare other ways to insert objects into a map
    m1.insert ( map <int, int> :: value_type ( 2, 20 ) );
    m1.insert ( cInt2Int ( 3, 30 ) );

    cout << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;

    // If the key already exists, operator[]
    // changes the value of the datum in the element
    m1[ 2 ] = 40;

    // operator[] will also insert the value of the data
    // type's default constructor if the value is unspecified
    m1[5];

    cout << "The keys of the mapped elements are now:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are now:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;

    // insert by moving key
    map<string, int> c2;
    string str("abc");
    cout << "c2[move(str)] == " << c2[move(str)] << endl;
    cout << "c2[\"abc\"] == " << c2["abc"] << endl;

    return (0);
}

```

The keys of the mapped elements are: 1 2 3.
 The values of the mapped elements are: 10 20 30.
 The keys of the mapped elements are now: 1 2 3 5.
 The values of the mapped elements are now: 10 40 30 0.
 c2[move(str)] == 0
 c2["abc"] == 1

map::operator=

Replaces the elements of a map with a copy of another map.

```
map& operator=(const map& right);

map& operator=(map&& right);
```

Parameters

PARAMETER	DESCRIPTION
Parameter	Description
<i>right</i>	The map being copied into the <code>map</code> .

Remarks

After erasing any existing elements in a `map`, `operator=` either copies or moves the contents of *right* into the map.

Example

```
// map_operator_as.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map<int, int> v1, v2, v3;
    map<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}
```

map::pointer

A type that provides a pointer to an element in a map.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a map object.

map::rbegin

Returns an iterator addressing the first element in a reversed map.

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed map.

Remarks

`rbegin` is used with a reversed map just as [begin](#) is used with a map.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the map object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the map object can be modified.

`rbegin` can be used to iterate through a map backwards.

Example

```

// map_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;

    map <int, int> :: iterator m1_Iter;
    map <int, int> :: reverse_iterator m1_rIter;
    map <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_rIter = m1.rbegin( );
    cout << "The first element of the reversed map m1 is "
         << m1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a map in a forward order
    cout << "The map is: ";
    for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
        cout << m1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a map in a reverse order
    cout << "The reversed map is: ";
    for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
        cout << m1_rIter -> first << " ";
    cout << "." << endl;

    // A map element can be erased by dereferencing to its key
    m1_rIter = m1.rbegin( );
    m1.erase ( m1_rIter -> first );

    m1_rIter = m1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed map is "
         << m1_rIter -> first << "." << endl;
}

```

```

The first element of the reversed map m1 is 3.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the first element in the reversed map is 2.

```

map::reference

A type that provides a reference to an element stored in a map.

```

typedef typename allocator_type::reference reference;

```

Example


```

// map_reference.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( m1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the map is "
         << Ref2 << "." << endl;

    //The non-const_reference can be used to modify the
    //data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}

```

```

The key of first element in the map is 1.
The data value of first element in the map is 10.
The modified data value of first element is 15.

```

map::rend

Returns an iterator that addresses the location succeeding the last element in a reversed map.

```

const_reverse_iterator rend() const;

reverse_iterator rend();

```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed map).

Remarks

`rend` is used with a reversed map just as `end` is used with a map.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the map object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the map object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its map.

The value returned by `rend` should not be dereferenced.

Example

```
// map_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;

    map <int, int> :: iterator m1_Iter;
    map <int, int> :: reverse_iterator m1_rIter;
    map <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "The last element of the reversed map m1 is "
         << m1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a map in a forward order
    cout << "The map is: ";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++)
        cout << m1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a map in a reverse order
    cout << "The reversed map is: ";
    for ( m1_rIter = m1.rbegin( ); m1_rIter != m1.rend( ); m1_rIter++)
        cout << m1_rIter -> first << " ";
    cout << "." << endl;

    // A map element can be erased by dereferencing to its key
    m1_rIter = --m1.rend( );
    m1.erase ( m1_rIter -> first );

    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed map is "
         << m1_rIter -> first << "." << endl;
}
```

```
The last element of the reversed map m1 is 1.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the last element in the reversed map is 2.
```

map::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed map.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` cannot modify the value of an element and is used to iterate through the map in reverse.

The `reverse_iterator` defined by map points to elements that are objects of [value_type](#), that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `reverse_iterator` *rIter* pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `rIter -> first`, which is equivalent to `(* rIter).first`. To access the value of the mapped datum for the element, use `rIter -> second`, which is equivalent to `(* rIter).second`.

Example

See example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

map::size

Returns the number of elements in the map.

```
size_type size() const;
```

Return Value

The current length of the map.

Example

The following example demonstrates the use of the `map::size` member function.

```
// map_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1, m2;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The map length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
    cout << "The map length is now " << i << "." << endl;
}
```

```
The map length is 1.
The map length is now 2.
```

map::size_type

An unsigned integer type that can represent the number of elements in a map.

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for [size](#) for an example of how to declare and use `size_type`.

map::swap

Exchanges the elements of two maps.

```
void swap(  
    map<Key, Type, Traits, Allocator>& right);
```

Parameters

right

The argument map providing the elements to be swapped with the target map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two maps whose elements are being exchanged.

Example

```

// map_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1, m2, m3;
    map <int, int>::iterator m1_Iter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );
    m2.insert ( Int_Pair ( 10, 100 ) );
    m2.insert ( Int_Pair ( 20, 200 ) );
    m3.insert ( Int_Pair ( 30, 300 ) );

    cout << "The original map m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter -> second;
    cout << "." << endl;

    // This is the member function version of swap
    //m2 is said to be the argument map; m1 the target map
    m1.swap( m2 );

    cout << "After swapping with m2, map m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter -> second;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( m1, m3 );

    cout << "After swapping with m3, map m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter -> second;
    cout << "." << endl;
}

```

```

The original map m1 is: 10 20 30.
After swapping with m2, map m1 is: 100 200.
After swapping with m3, map m1 is: 300.

```

map::upper_bound

Returns an iterator to the first element in a map that with a key having a value that is greater than that of a specified key.

```

iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;

```

Parameters

key

The argument key value to be compared with the sort key value of an element from the map being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value is assigned to a `const_iterator`, the map object cannot be modified. If the return value is assigned to a `iterator`, the map object can be modified.

Example

```
// map_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    map <int, int> m1;
    map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_RcIter = m1.upper_bound( 2 );
    cout << "The first element of map m1 with a key "
         << "greater than 2 is: "
         << m1_RcIter -> second << "." << endl;

    // If no match is found for the key, end is returned
    m1_RcIter = m1.upper_bound ( 4 );

    if ( m1_RcIter == m1.end( ) )
        cout << "The map m1 doesn't have an element "
             << "with a key greater than 4." << endl;
    else
        cout << "The element of map m1 with a key > 4 is: "
             << m1_RcIter -> second << "." << endl;

    // The element at a specific location in the map can be found
    // using a dereferenced iterator addressing the location
    m1_AcIter = m1.begin( );
    m1_RcIter = m1.upper_bound ( m1_AcIter -> first );
    cout << "The 1st element of m1 with a key greater than\n"
         << "that of the initial element of m1 is: "
         << m1_RcIter -> second << "." << endl;
}
```

```
The first element of map m1 with a key greater than 2 is: 30.
The map m1 doesn't have an element with a key greater than 4.
The 1st element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

map::value_comp

The member function returns a function object that determines the order of elements in a map by comparing their key values.

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a map uses to order its elements.

Remarks

For a map m , if two elements $e1(k1, d1)$ and $e2(k2, d2)$ are objects of type `value_type`, where $k1$ and $k2$ are their keys of type `key_type` and $d1$ and $d2$ are their data of type `mapped_type`, then `m.value_comp(e1, e2)` is equivalent to `m.key_comp(k1, k2)`. A stored object defines the member function

```
bool operator( value_type& left, value_type& right);
```

which returns **true** if the key value of `left` precedes and is not equal to the key value of `right` in the sort order.

Example

```
// map_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;

    map <int, int, less<int> > m1;
    map <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
    pair< map<int,int>::iterator, bool > pr1, pr2;

    pr1= m1.insert ( map <int, int> :: value_type ( 1, 10 ) );
    pr2= m1.insert ( map <int, int> :: value_type ( 2, 5 ) );

    if( vc1( *pr1.first, *pr2.first ) == true )
    {
        cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
              << endl;
    }
    else
    {
        cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
              << endl;
    }

    if(vc1( *pr2.first, *pr1.first ) == true )
    {
        cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
              << endl;
    }
    else
    {
        cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
              << endl;
    }
}
```

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

map::value_type

The type of object stored as an element in a map.

```
typedef pair<const Key, Type> value_type;
```

Example

```
// map_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    typedef pair <const int, int> cInt2Int;
    map <int, int> m1;
    map <int, int> :: key_type key1;
    map <int, int> :: mapped_type mapped1;
    map <int, int> :: value_type value1;
    map <int, int> :: iterator pIter;

    // value_type can be used to pass the correct type
    // explicitly to avoid implicit type conversion
    m1.insert ( map <int, int> :: value_type ( 1, 10 ) );

    // Compare other ways to insert objects into a map
    m1.insert ( cInt2Int ( 2, 20 ) );
    m1[ 3 ] = 30;

    // Initializing key1 and mapped1
    key1 = ( m1.begin( ) -> first );
    mapped1 = ( m1.begin( ) -> second );

    cout << "The key of first element in the map is "
          << key1 << "." << endl;

    cout << "The data value of first element in the map is "
          << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

See also

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

multimap Class

11/14/2018 • 53 minutes to read • [Edit Online](#)

The C++ Standard Library multimap class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key. The value of the key does not need to be unique and is used to order the data automatically. The value of an element in a multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

Syntax

```
template <class Key,  
          class Type,  
          class Traits=less <Key>,  
          class Allocator=allocator <pair <const Key, Type>>>  
class multimap;
```

Parameters

Key

The key data type to be stored in the multimap.

Type

The element data type to be stored in the multimap.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multimap. The binary predicate `less<Key>` is the default value.

In C++ 14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Allocator

The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is `allocator<pair <const Key, Type> >`.

Remarks

The C++ Standard Library multimap class is

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of

data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the class member functions [insert](#) and [multimap](#) have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The multimap should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words were not always uniquely defined. If, instead, the key words were uniquely defined so that keys were unique, then a map would be the container of choice. If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure.

The multimap orders the sequence it controls by calling a stored function object of type [key_compare](#). This stored object is a comparison function that may be accessed by calling the member function [key_comp](#). In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate `f(x,y)` is a function object that has two argument objects `x` and `y` and a return value of true or false. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects `x` and `y` are defined to be equivalent when both `f(x,y)` and `f(y,x)` are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Members

Constructors

CONSTRUCTOR	DESCRIPTION
multimap	Constructs a <code>multimap</code> that is empty or that is a copy of all or part of some other <code>multimap</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for the <code>multimap</code> object.
<code>const_iterator</code>	A type that provides a bidirectional iterator that can read a const element in the <code>multimap</code> .
<code>const_pointer</code>	A type that provides a pointer to a const element in a <code>multimap</code> .
<code>const_reference</code>	A type that provides a reference to a const element stored in a <code>multimap</code> for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in the <code>multimap</code> .
<code>difference_type</code>	A signed integer type that can be used to represent the number of elements of a <code>multimap</code> in a range between elements pointed to by iterators.
<code>iterator</code>	A type that provides the difference between two iterators that refer to elements within the same <code>multimap</code> .
<code>key_compare</code>	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>multimap</code> .
<code>key_type</code>	A type that describes the sort key object that constitutes each element of the <code>multimap</code> .
<code>mapped_type</code>	A type that represents the data type stored in a <code>multimap</code> .
<code>pointer</code>	A type that provides a pointer to a const element in a <code>multimap</code> .
<code>reference</code>	A type that provides a reference to an element stored in a <code>multimap</code> .
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed <code>multimap</code> .
<code>size_type</code>	An unsigned integer type that provides a pointer to a const element in a <code>multimap</code> .
<code>value_type</code>	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the <code>multimap</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
-----------------	-------------

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns an iterator addressing the first element in the <code>multimap</code> .
<code>cbegin</code>	Returns a const iterator addressing the first element in the <code>multimap</code> .
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a <code>multimap</code> .
<code>clear</code>	Erases all the elements of a <code>multimap</code> .
<code>count</code>	Returns the number of elements in a <code>multimap</code> whose key matches a parameter-specified key.
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed <code>multimap</code> .
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed <code>multimap</code> .
<code>emplace</code>	Inserts an element constructed in place into a <code>multimap</code> .
<code>emplace_hint</code>	Inserts an element constructed in place into a <code>multimap</code> , with a placement hint
<code>empty</code>	Tests if a <code>multimap</code> is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a <code>multimap</code> .
<code>equal_range</code>	Finds the range of elements where the key of the element matches a specified value.
<code>erase</code>	Removes an element or a range of elements in a <code>multimap</code> from specified positions or removes elements that match a specified key.
<code>find</code>	Returns an iterator addressing the first location of an element in a <code>multimap</code> that has a key equivalent to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct the <code>multimap</code> .
<code>insert</code>	Inserts an element or a range of elements into a <code>multimap</code> .
<code>key_comp</code>	Retrieves a copy of the comparison object used to order keys in a <code>multimap</code> .
<code>lower_bound</code>	Returns an iterator to the first element in a <code>multimap</code> that with a key that is equal to or greater than a specified key.

MEMBER FUNCTION	DESCRIPTION
max_size	Returns the maximum length of the <code>multimap</code> .
rbegin	Returns an iterator addressing the first element in a reversed <code>multimap</code> .
rend	Returns an iterator that addresses the location succeeding the last element in a reversed <code>multimap</code> .
size	Returns the number of elements in the <code>multimap</code> .
swap	Exchanges the elements of two <code>multimap</code> s.
upper_bound	Returns an iterator to the first element in a <code>multimap</code> that with a key that is greater than a specified key.
value_comp	The member function returns a function object that determines the order of elements in a <code>multimap</code> by comparing their key values.

Operators

OPERATOR	DESCRIPTION
operator=	Replaces the elements of a <code>multimap</code> with a copy of another <code>multimap</code> .

Requirements

Header: `<map>`

Namespace: `std`

The (**key**, **value**) pairs are stored in a `multimap` as objects of type `pair`. The `pair` class requires the header `<utility>`, which is automatically included by `<map>`.

`multimap::allocator_type`

A type that represents the allocator class for the `multimap` object.

```
typedef Allocator allocator_type;
```

Example

See the example for [get_allocator](#) for an example using `allocator_type`.

`multimap::begin`

Returns an iterator addressing the first element in the `multimap`.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the multimap or the location succeeding an empty multimap.

Example

```
// multimap_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;

    multimap <int, int> :: iterator m1_Iter;
    multimap <int, int> :: const_iterator m1_cIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 0, 0 ) );
    m1.insert ( Int_Pair ( 1, 1 ) );
    m1.insert ( Int_Pair ( 2, 4 ) );

    m1_cIter = m1.begin ( );
    cout << "The first element of m1 is " << m1_cIter -> first << endl;

    m1_Iter = m1.begin ( );
    m1.erase ( m1_Iter );

    // The following 2 lines would err as the iterator is const
    // m1_cIter = m1.begin ( );
    // m1.erase ( m1_cIter );

    m1_cIter = m1.begin( );
    cout << "First element of m1 is now " << m1_cIter -> first << endl;
}
```

```
The first element of m1 is 0
First element of m1 is now 1
```

multimap::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`multimap::cend`

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

`multimap::clear`

Erases all the elements of a multimap.

```
void clear();
```

Example

The following example demonstrates the use of the `multimap::clear` member function.

```
// multimap_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> m1;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 4));

    i = m1.size();
    cout << "The size of the multimap is initially "
         << i << "." << endl;

    m1.clear();
    i = m1.size();
    cout << "The size of the multimap after clearing is "
         << i << "." << endl;
}
```

```
The size of the multimap is initially 2.
The size of the multimap after clearing is 0.
```

multimap::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the multimap.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by multimap points to objects of [value_type](#), which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_iterator` *cIter* pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See the example for [begin](#) for an example using `const_iterator`.

multimap::const_pointer

A type that provides a pointer to a **const** element in a multimap.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a multimap object.

multimap::const_reference

A type that provides a reference to a **const** element stored in a multimap for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// multimap_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( m1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of the first element in the multimap is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of the first element in the multimap is "
         << Ref2 << "." << endl;
}
```

```
The key of the first element in the multimap is 1.
The data value of the first element in the multimap is 10.
```

multimap::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the multimap.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the multimap in reverse.

The `const_reverse_iterator` defined by `multimap` points to objects of `value_type`, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_reverse_iterator` *crIter* pointing to an element in a `multimap`, use the `->` operator.

To access the value of the key for the element, use `crIter->first`, which is equivalent to `(*crIter).first`. To access the value of the mapped datum for the element, use `crIter->second`, which is equivalent to `(*crIter).second`.

Example

See the example for [rend](#) for an example of how to declare and use `const_reverse_iterator`.

multimap::count

Returns the number of elements in a `multimap` whose keys match a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the `multimap`.

Return Value

The number of elements whose sort keys match the parameter `key`; 0 if the `multimap` doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

`[lower_bound(key), upper_bound(key))`

that have a key value *key*.

Example

The following example demonstrates the use of the `multimap::count` member function.

```

// multimap_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> m1;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Elements do not need to have unique keys in multimap,
    // so duplicates are allowed and counted
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}

```

```

The number of elements in m1 with a sort key of 1 is: 2.
The number of elements in m1 with a sort key of 2 is: 2.
The number of elements in m1 with a sort key of 3 is: 0.

```

multimap::crbegin

Returns a const iterator addressing the first element in a reversed multimap.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed [multimap](#) or addressing what had been the last element in the unreversed `multimap`.

Remarks

`crbegin` is used with a reversed `multimap` just as [begin](#) is used with a `multimap`.

With the return value of `crbegin`, the `multimap` object cannot be modified.

`crbegin` can be used to iterate through a `multimap` backwards.

Example

```

// multimap_crbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;

    multimap <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_crIter = m1.crbegin( );
    cout << "The first element of the reversed multimap m1 is "
         << m1_crIter -> first << "." << endl;
}

```

The first element of the reversed multimap m1 is 3.

multimap::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed multimap.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed [multimap](#) (the location that had preceded the first element in the unreversed `multimap`).

Remarks

`crend` is used with a reversed `multimap` just as [multimap::end](#) is used with a `multimap`.

With the return value of `crend`, the `multimap` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `multimap`.

The value returned by `crend` should not be dereferenced.

Example

```

// multimap_crend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;

    multimap <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_crIter = m1.crend( );
    m1_crIter--;
    cout << "The last element of the reversed multimap m1 is "
         << m1_crIter -> first << "." << endl;
}

```

The last element of the reversed multimap m1 is 1.

multimap::difference_type

A signed integer type that can be used to represent the number of elements of a multimap in a range between elements pointed to by iterators.

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range *[first, last)* between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

```

// multimap_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 3, 20 ) );

    // The following will insert as multimap keys are not unique
    m1.insert ( Int_Pair ( 2, 30 ) );

    multimap <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
    m1_bIter = m1.begin( );
    m1_eIter = m1.end( );

    // Count the number of elements in a multimap
    multimap <int, int>::difference_type df_count = 0;
    m1_Iter = m1.begin( );
    while ( m1_Iter != m1_eIter )
    {
        df_count++;
        m1_Iter++;
    }

    cout << "The number of elements in the multimap m1 is: "
         << df_count << "." << endl;
}

```

```
The number of elements in the multimap m1 is: 4.
```

multimap::emplace

Inserts an element constructed in place (no copy or move operations are performed).

```

template <class... Args>
iterator emplace(Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the multimap.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

If an exception is thrown during the insertion, the container is left unaltered and the exception is rethrown.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

```
// multimap_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: " << endl;

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    multimap<string, string> m1;

    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "multimap modified, now contains ";
    print(m1);
    cout << endl;

    m1.emplace("Bob", "Engineering");

    cout << "multimap modified, now contains ";
    print(m1);
    cout << endl;
}
```

multimap::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the multimap.

PARAMETER	DESCRIPTION
<i>where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>where</i> , insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state is not modified.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see [map::emplace_hint](#).

multimap::empty

Tests if a multimap is empty.

```
bool empty() const;
```

Return Value

true if the multimap is empty; **false** if the multimap is nonempty.

Example

```
// multimap_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1, m2;

    typedef pair <int, int> Int_Pair;
    m1.insert ( Int_Pair ( 1, 1 ) );

    if ( m1.empty( ) )
        cout << "The multimap m1 is empty." << endl;
    else
        cout << "The multimap m1 is not empty." << endl;

    if ( m2.empty( ) )
        cout << "The multimap m2 is empty." << endl;
    else
        cout << "The multimap m2 is not empty." << endl;
}
```

```
The multimap m1 is not empty.
The multimap m2 is empty.
```


multimap::end

Returns the past-the-end iterator.

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the multimap is empty, then `multimap::end() == multimap::begin()`.

Remarks

end is used to test whether an iterator has passed the end of its multimap.

The value returned by **end** should not be dereferenced.

For a code example, see [multimap::find](#).

multimap::equal_range

Finds the range of elements where the key of the element matches a specified value.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

A pair of iterators such that the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```

// multimap_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    typedef multimap <int, int, less<int> > IntMMap;
    IntMMap m1;
    multimap <int, int> :: const_iterator m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
    p1 = m1.equal_range( 2 );

    cout << "The lower bound of the element with "
         << "a key of 2 in the multimap m1 is: "
         << p1.first -> second << "." << endl;

    cout << "The upper bound of the element with "
         << "a key of 2 in the multimap m1 is: "
         << p1.second -> second << "." << endl;

    // Compare the upper_bound called directly
    m1_RcIter = m1.upper_bound( 2 );

    cout << "A direct call of upper_bound( 2 ) gives "
         << m1_RcIter -> second << "," << endl
         << "matching the 2nd element of the pair "
         << "returned by equal_range( 2 )." << endl;

    p2 = m1.equal_range( 4 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
        cout << "The multimap m1 doesn't have an element "
             << "with a key less than 4." << endl;
    else
        cout << "The element of multimap m1 with a key >= 40 is: "
             << p1.first -> first << "." << endl;
}

```

```

The lower bound of the element with a key of 2 in the multimap m1 is: 20.
The upper bound of the element with a key of 2 in the multimap m1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 2 ).
The multimap m1 doesn't have an element with a key less than 4.

```

multimap::erase

Removes an element or a range of elements in a multimap from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the multimap.

Remarks

For a code example, see [map::erase](#).

multimap::find

Returns an iterator that refers to the first location of an element in a multimap that has a key equivalent to a specified key.

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key

The key value to be matched by the sort key of an element from the multimap being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the multimap (`multimap::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the multimap whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the multimap object cannot be modified. If the return value of `find` is assigned to an `iterator`, the multimap object can be modified.

Example

```
// compile with: /EHsc /W4 /MTd
#include <map>
#include <iostream>
#include <vector>
#include <string>
#include <utility> // make_pair()

using namespace std;

template <typename A, typename B> void print_elem(const pair<A, B>& p) {
    cout << "(" << p.first << ", " << p.second << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    multimap<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting multimap m1 is (key, value):" << endl;
    print_collection(m1);

    vector<pair<int, string>> v;
    v.push_back(make_pair(43, "Tc"));
    v.push_back(make_pair(41, "Nb"));
    v.push_back(make_pair(46, "Pd"));
    v.push_back(make_pair(42, "Mo"));
    v.push_back(make_pair(44, "Ru"));
    v.push_back(make_pair(44, "Ru")); // attempt a duplicate

    cout << "Inserting the following vector data into m1:" << endl;
    print_collection(v);

    m1.insert(v.begin(), v.end());

    cout << "The modified multimap m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
    findit(m1, 6);
}
```

multimap::get_allocator

Returns a copy of the allocator object used to construct the multimap.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the multimap.

Remarks

Allocators for the multimap class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// multimap_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int>::allocator_type m1_Alloc;
    multimap <int, int>::allocator_type m2_Alloc;
    multimap <int, double>::allocator_type m3_Alloc;
    multimap <int, int>::allocator_type m4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    multimap <int, int> m1;
    multimap <int, int, allocator<int> > m2;
    multimap <int, double, allocator<double> > m3;

    m1_Alloc = m1.get_allocator( );
    m2_Alloc = m2.get_allocator( );
    m3_Alloc = m3.get_allocator( );

    cout << "The number of integers that can be allocated"
         << endl << "before free memory is exhausted: "
         << m2.max_size( ) << ".\n" << endl;

    cout << "The number of doubles that can be allocated"
         << endl << "before free memory is exhausted: "
         << m3.max_size( ) << ".\n" << endl;

    // The following line creates a multimap m4
    // with the allocator of multimap m1.
    map <int, int> m4( less<int>( ), m1_Alloc );

    m4_Alloc = m4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated via the other
    if( m1_Alloc == m4_Alloc )
    {
        cout << "The allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "The allocators are not interchangeable."
             << endl;
    }
}
```

multimap::insert

Inserts an element or a range of elements into a multimap.

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);
```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the multimap.
<i>Where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>Where</i> , insertion can occur in amortized constant time instead of logarithmic time.)
<i>ValTy</i>	Template parameter that specifies the argument type that the map can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the multimap.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the multimap.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The [value_type](#) of a container is a typedef that belongs to the container, and for map, `multimap<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into a multimap that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *Last* does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [multimap::emplace](#) and [multimap::emplace_hint](#).

Example

```
// multimap_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>
#include <vector>
#include <utility> // make_pair()

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    // insert single values
    multimap<int, int> m1;
    // call insert(const value_type&) version
    m1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    m1.insert(make_pair(2, 20));

    cout << "The original key and mapped values of m1 are:" << endl;
    print(m1);

    // intentionally attempt a duplicate, single element
    m1.insert(make_pair(1, 11));
}
```

```

cout << "The modified key and mapped values of m1 are:" << endl;
print(m1);

// single element, with hint
m1.insert(m1.end(), make_pair(3, 30));
cout << "The modified key and mapped values of m1 are:" << endl;
print(m1);
cout << endl;

// The templated version inserting a jumbled range
multimap<int, int> m2;
vector<pair<int, int>> v;
v.push_back(make_pair(43, 294));
v.push_back(make_pair(41, 262));
v.push_back(make_pair(45, 330));
v.push_back(make_pair(42, 277));
v.push_back(make_pair(44, 311));

cout << "Inserting the following vector data into m2:" << endl;
print(v);

m2.insert(v.begin(), v.end());

cout << "The modified key and mapped values of m2 are:" << endl;
print(m2);
cout << endl;

// The templated versions move-constructing elements
multimap<int, string> m3;
pair<int, string> ip1(475, "blue"), ip2(510, "green");

// single element
m3.insert(move(ip1));
cout << "After the first move insertion, m3 contains:" << endl;
print(m3);

// single element with hint
m3.insert(m3.end(), move(ip2));
cout << "After the second move insertion, m3 contains:" << endl;
print(m3);
cout << endl;

multimap<int, int> m4;
// Insert the elements from an initializer_list
m4.insert({ { 4, 44 }, { 2, 22 }, { 3, 33 }, { 1, 11 }, { 5, 55 } });
cout << "After initializer_list insertion, m4 contains:" << endl;
print(m4);
cout << endl;
}

```

multimap::iterator

A type that provides a bidirectional iterator that can read or modify any element in a multimap.

```
typedef implementation-defined iterator;
```

Remarks

The `iterator` defined by `multimap` points to objects of `value_type`, which are of type `pair<const Key, Type>`. The value of the key is available through the first member `pair` and the value of the mapped element is available through the second member of the `pair`.

To dereference an `iterator` *Iter* pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).second`.

A type `iterator` can be used to modify the value of an element.

Example

See the example for [begin](#) for an example of how to declare and use `iterator`.

multimap::key_comp

Retrieves a copy of the comparison object used to order keys in a multimap.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a multimap uses to order its elements.

Remarks

The stored object defines the member function

```
bool operator( const Key& x, const Key& y);
```

which returns true if *x* strictly precedes *y* in the sort order.

Example

```

// multimap_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;

    multimap <int, int, less<int> > m1;
    multimap <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
              << "where kc1 is the function object of m1."
              << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
              << "where kc1 is the function object of m1."
              << endl;
    }

    multimap <int, int, greater<int> > m2;
    multimap <int, int, greater<int> >::key_compare kc2 = m2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true, "
              << "where kc2 is the function object of m2."
              << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
              << "where kc2 is the function object of m2."
              << endl;
    }
}

```

```

kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.

```

multimap::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the multimap.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the [multimap Class](#) topic.

Example

See the example for [key_comp](#) for an example of how to declare and use `key_compare`.

multimap::key_type

A type that describes the sort key object that constitutes each element of the multimap.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the [multimap Class](#) topic.

Example

See the example for [value_type](#) for an example of how to declare and use `key_type`.

multimap::lower_bound

Returns an iterator to the first element in a multimap that with a key that is equal to or greater than a specified key.

```
iterator lower_bound(const Key& key);  
  
const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a multimap that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the multimap if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the multimap object cannot be modified. If the return value of `lower_bound` is assigned to an **iterator**, the multimap object can be modified.

Example

```

// multimap_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_RcIter = m1.lower_bound( 2 );
    cout << "The element of multimap m1 with a key of 2 is: "
         << m1_RcIter -> second << "." << endl;

    m1_RcIter = m1.lower_bound( 3 );
    cout << "The first element of multimap m1 with a key of 3 is: "
         << m1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    m1_RcIter = m1.lower_bound( 4 );

    if ( m1_RcIter == m1.end( ) )
        cout << "The multimap m1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of multimap m1 with a key of 4 is: "
             << m1_RcIter -> second << "." << endl;

    // The element at a specific location in the multimap can be
    // found using a dereferenced iterator addressing the location
    m1_AcIter = m1.end( );
    m1_AcIter--;
    m1_RcIter = m1.lower_bound( m1_AcIter -> first );
    cout << "The first element of m1 with a key matching\n"
         << "that of the last element is: "
         << m1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if ( m1_RcIter == --m1.end( ) )
        cout << "This is the last element of multimap m1."
             << endl;
    else
        cout << "This is not the last element of multimap m1."
             << endl;
}

```

```

The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key matching
that of the last element is: 20.
This is not the last element of multimap m1.

```

multimap::mapped_type

A type that represents the data type stored in a multimap.

```
typedef Type mapped_type;
```

Remarks

`mapped_type` is a synonym for the template parameter `Type` .

For more information on `Type` see the [multimap Class](#) topic.

Example

See the example for [value_type](#) for an example of how to declare and use `key_type` .

multimap::max_size

Returns the maximum length of the multimap.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the multimap.

Example

```
// multimap_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    multimap <int, int> :: size_type i;

    i = m1.max_size( );
    cout << "The maximum possible length "
         << "of the multimap is " << i << "." << endl;
}
```

multimap::multimap

Constructs a multimap that is empty or that is a copy of all or part of some other multimap.

```

multimap();

explicit multimap(
    const Traits& Comp);

multimap(
    const Traits& Comp,
    const Allocator& Al);

map(
    const multimap& Right);

multimap(
    multimap&& Right);

multimap(
    initializer_list<value_type> IList);

multimap(
    initializer_list<value_type> IList,
    const Compare& Comp);

multimap(
    initializer_list<value_type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this multimap object, which defaults to Allocator.
<i>Comp</i>	The comparison function of type <code>constTraits</code> used to order the elements in the map, which defaults to <code>Traits</code> .
<i>Right</i>	The map of which the constructed set is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the multimap and that can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their multimap.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the multimap and that can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial multimap, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the multimap *Right*.

The fifth constructor specifies a copy of the multimap by moving *Right*.

The sixth, seventh, and eighth constructors copy the members of an initializer_list.

The next three constructors copy the range `[First, Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

Example

```
// multimap_ctor.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    typedef pair <int, int> Int_Pair;

    // Create an empty multimap m0 of key type integer
    multimap <int, int> m0;

    // Create an empty multimap m1 with the key comparison
    // function of less than, then insert 4 elements
    multimap <int, int, less<int> > m1;
    m1.insert(Int_Pair(1, 10));
    m1.insert(Int_Pair(2, 20));
    m1.insert(Int_Pair(3, 30));
    m1.insert(Int_Pair(4, 40));

    // Create an empty multimap m2 with the key comparison
    // function of greater than, then insert 2 elements
    multimap <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a multimap m3 with the
    // allocator of multimap m1
    multimap <int, int>::allocator_type m1_Alloc;
    m1_Alloc = m1.get_allocator();
    multimap <int, int> m3(less<int>(), m1_Alloc);
    m3.insert(Int_Pair(3, 30));
```

```

// Create a copy, multimap m4, of multimap m1
multimap<int, int> m4(m1);

// Create a multimap m5 by copying the range m1[ first, last)
multimap<int, int>::const_iterator m1_bcIter, m1_ecIter;
m1_bcIter = m1.begin();
m1_ecIter = m1.begin();
m1_ecIter++;
m1_ecIter++;
multimap<int, int> m5(m1_bcIter, m1_ecIter);

// Create a multimap m6 by copying the range m4[ first, last)
// and with the allocator of multimap m2
multimap<int, int>::allocator_type m2_Alloc;
m2_Alloc = m2.get_allocator();
multimap<int, int> m6(m4.begin(), ++m4.begin(), less<int>(), m2_Alloc);

cout << "m1 =";
for (auto i : m1)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m2 =";
for (auto i : m2)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m3 =";
for (auto i : m3)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m4 =";
for (auto i : m4)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m5 =";
for (auto i : m5)
    cout << i.first << " " << i.second << ", ";
cout << endl;

cout << "m6 =";
for (auto i : m6)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a multimap m8 by copying in an initializer_list
multimap<int, int> m8{ { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 } };
cout << "m8: = ";
for (auto i : m8)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a multimap m9 with an initializer_list and a comparator
multimap<int, int> m9({ { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 } }, less<int>());
cout << "m9: = ";
for (auto i : m9)
    cout << i.first << " " << i.second << ", ";
cout << endl;

// Create a multimap m10 with an initializer_list, a comparator, and an allocator
multimap<int, int> m10({ { 9, 9 }, { 10, 10 }, { 11, 11 }, { 12, 12 } }, less<int>(),
m9.get_allocator());
cout << "m10: = ";
for (auto i : m10)
    cout << i.first << " " << i.second << ", ";
cout << endl;

```



```
}
```

multimap::operator=

Replaces the elements of a multimap with a copy of another multimap.

```
multimap& operator=(const multimap& right);

multimap& operator=(multimap&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The multimap being copied into the <code>multimap</code> .

Remarks

After erasing any existing elements in a `multimap`, `operator=` either copies or moves the contents of *right* into the `multimap`.

Example

```
// multimap_operator_as.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> v1, v2, v3;
    multimap<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}
```

multimap::pointer

A type that provides a pointer to an element in a multimap.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a multimap object.

multimap::rbegin

Returns an iterator addressing the first element in a reversed multimap.

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed multimap or addressing what had been the last element in the unreversed multimap.

Remarks

`rbegin` is used with a reversed multimap just as [begin](#) is used with a multimap.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the multimap object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the multimap object can be modified.

`rbegin` can be used to iterate through a multimap backwards.

Example

```

// multimap_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;

    multimap <int, int> :: iterator m1_Iter;
    multimap <int, int> :: reverse_iterator m1_rIter;
    multimap <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_rIter = m1.rbegin( );
    cout << "The first element of the reversed multimap m1 is "
         << m1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a multimap in a forward order
    cout << "The multimap is: ";
    for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
        cout << m1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a multimap in a reverse order
    cout << "The reversed multimap is: ";
    for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
        cout << m1_rIter -> first << " ";
    cout << "." << endl;

    // A multimap element can be erased by dereferencing its key
    m1_rIter = m1.rbegin( );
    m1.erase ( m1_rIter -> first );

    m1_rIter = m1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed multimap is "
         << m1_rIter -> first << "." << endl;
}

```

```

The first element of the reversed multimap m1 is 3.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the first element in the reversed multimap is 2.

```

multimap::reference

A type that provides a reference to an element stored in a multimap.

```

typedef typename allocator_type::reference reference;

```

Example

```

// multimap_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );

    // Declare and initialize a const_reference &Ref1
    // to the key of the first element
    const int &Ref1 = ( m1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the multimap is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the multimap is "
         << Ref2 << "." << endl;

    // The non-const_reference can be used to modify the
    // data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}

```

```

The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The modified data value of first element is 15.

```

multimap::rend

Returns an iterator that addresses the location succeeding the last element in a reversed multimap.

```

const_reverse_iterator rend() const;

reverse_iterator rend();

```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multimap (the location that had preceded the first element in the unreversed multimap).

Remarks

`rend` is used with a reversed multimap just as `end` is used with a multimap.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the multimap object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the multimap object can be

modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its multimap.

The value returned by `rend` should not be dereferenced.

Example

```
// multimap_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;

    multimap <int, int> :: iterator m1_Iter;
    multimap <int, int> :: reverse_iterator m1_rIter;
    multimap <int, int> :: const_reverse_iterator m1_crIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );

    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "The last element of the reversed multimap m1 is "
         << m1_rIter -> first << "." << endl;

    // begin can be used to start an iteration
    // through a multimap in a forward order
    cout << "The multimap is: ";
    for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ) ; m1_Iter++)
        cout << m1_Iter -> first << " ";
    cout << "." << endl;

    // rbegin can be used to start an iteration
    // through a multimap in a reverse order
    cout << "The reversed multimap is: ";
    for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ) ; m1_rIter++)
        cout << m1_rIter -> first << " ";
    cout << "." << endl;

    // A multimap element can be erased by dereferencing to its key
    m1_rIter = --m1.rend( );
    m1.erase ( m1_rIter -> first );

    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed multimap is "
         << m1_rIter -> first << "." << endl;
}
```

```
The last element of the reversed multimap m1 is 1.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the last element in the reversed multimap is 2.
```

multimap::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed multimap.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the multimap in reverse.

The `reverse_iterator` defined by multimap points to objects of [value_type](#), which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `reverse_iterator` *rIter* pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `rIter->first`, which is equivalent to `(*rIter).first`. To access the value of the mapped datum for the element, use `rIter->second`, which is equivalent to `(*rIter).second`.

Example

See the example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

multimap::size

Returns the number of elements in the multimap.

```
size_type size() const;
```

Return Value

The current length of the multimap.

Example

The following example demonstrates the use of the `multimap::size` member function.

```
// multimap_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    multimap<int, int> m1, m2;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The multimap length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
    cout << "The multimap length is now " << i << "." << endl;
}
```

```
The multimap length is 1.
The multimap length is now 2.
```

multimap::size_type

An unsigned integer type that counts the number of elements in a multimap.

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for [size](#) for an example of how to declare and use `size_type`

multimap::swap

Exchanges the elements of two multimaps.

```
void swap(  
    multimap<Key, Type, Traits, Allocator>& right);
```

Parameters

right

The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap `left`.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two multimaps whose elements are being exchanged.

Example

```

// multimap_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1, m2, m3;
    multimap <int, int>::iterator m1_Iter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );
    m2.insert ( Int_Pair ( 10, 100 ) );
    m2.insert ( Int_Pair ( 20, 200 ) );
    m3.insert ( Int_Pair ( 30, 300 ) );

    cout << "The original multimap m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter->second;
    cout << "." << endl;

    // This is the member function version of swap
    m1.swap( m2 );

    cout << "After swapping with m2, multimap m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter->second;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( m1, m3 );

    cout << "After swapping with m3, multimap m1 is:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " " << m1_Iter->second;
    cout << "." << endl;
}

```

The original multimap m1 is: 10 20 30.
 After swapping with m2, multimap m1 is: 100 200.
 After swapping with m3, multimap m1 is: 300.

multimap::upper_bound

Returns an iterator to the first element in a multimap that with a key that is greater than a specified key.

```

iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;

```

Parameters

key

The argument key to be compared with the sort key of an element from the multimap being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a multimap that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the multimap if no

match is found for the key.

If the return value is assigned to a `const_iterator`, the multimap object cannot be modified. If the return value is assigned to a `iterator`, the multimap object can be modified.

Example

```
// multimap_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap <int, int> m1;
    multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
    typedef pair <int, int> Int_Pair;

    m1.insert ( Int_Pair ( 1, 10 ) );
    m1.insert ( Int_Pair ( 2, 20 ) );
    m1.insert ( Int_Pair ( 3, 30 ) );
    m1.insert ( Int_Pair ( 3, 40 ) );

    m1_RcIter = m1.upper_bound( 1 );
    cout << "The 1st element of multimap m1 with "
         << "a key greater than 1 is: "
         << m1_RcIter -> second << "." << endl;

    m1_RcIter = m1.upper_bound( 2 );
    cout << "The first element of multimap m1 with a key "
         << " greater than 2 is: "
         << m1_RcIter -> second << "." << endl;

    // If no match is found for the key, end( ) is returned
    m1_RcIter = m1.lower_bound( 4 );

    if ( m1_RcIter == m1.end( ) )
        cout << "The multimap m1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of multimap m1 with a key of 4 is: "
             << m1_RcIter -> second << "." << endl;

    // The element at a specific location in the multimap can be
    // found using a dereferenced iterator addressing the location
    m1_AcIter = m1.begin( );
    m1_RcIter = m1.upper_bound( m1_AcIter -> first );
    cout << "The first element of m1 with a key greater than\n"
         << "that of the initial element of m1 is: "
         << m1_RcIter -> second << "." << endl;
}
```

```
The 1st element of multimap m1 with a key greater than 1 is: 20.
The first element of multimap m1 with a key greater than 2 is: 30.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

multimap::value_comp

The member function returns a function object that determines the order of elements in a multimap by comparing their key values.

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a multimap uses to order its elements.

Remarks

For a multimap m , if two elements $e1(k1, d1)$ and $e2(k2, d2)$ are objects of type `value_type`, where $k1$ and $k2$ are their keys of type `key_type` and $d1$ and $d2$ are their data of type `mapped_type`, then `m.value_comp(e1, e2)` is equivalent to `m.key_comp(k1, k2)`.

Example

```
// multimap_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;

    multimap <int, int, less<int> > m1;
    multimap <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
    multimap<int,int>::iterator Iter1, Iter2;

    Iter1= m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );
    Iter2= m1.insert ( multimap <int, int> :: value_type ( 2, 5 ) );

    if( vc1( *Iter1, *Iter2 ) == true )
    {
        cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
              << endl;
    }
    else
    {
        cout << "The element ( 1,10 ) does "
              << "not precede the element ( 2,5 )."
              << endl;
    }

    if( vc1( *Iter2, *Iter1 ) == true )
    {
        cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
              << endl;
    }
    else
    {
        cout << "The element ( 2,5 ) does "
              << "not precede the element ( 1,10 )."
              << endl;
    }
}
```

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

multimap::value_type

A type that represents the type of object stored as an element in a map.

```
typedef pair<const Key, Type> value_type;
```

Example

```
// multimap_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    typedef pair <const int, int> cInt2Int;
    multimap <int, int> m1;
    multimap <int, int> :: key_type key1;
    multimap <int, int> :: mapped_type mapped1;
    multimap <int, int> :: value_type value1;
    multimap <int, int> :: iterator pIter;

    // value_type can be used to pass the correct type
    // explicitly to avoid implicit type conversion
    m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );

    // Compare another way to insert objects into a hash_multimap
    m1.insert ( cInt2Int ( 2, 20 ) );

    // Initializing key1 and mapped1
    key1 = ( m1.begin( ) -> first );
    mapped1 = ( m1.begin( ) -> second );

    cout << "The key of first element in the multimap is "
         << key1 << "." << endl;

    cout << "The data value of first element in the multimap is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.
```

See also

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

value_compare Class (<map>)

10/31/2018 • 2 minutes to read • [Edit Online](#)

Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.

Syntax

```
class value_compare : public binary_function<value_type, value_type, bool>
{
public:
    bool operator()(const value_type& left, const value_type& right) const;
    value_compare(key_compare pred) : comp(pred);
protected:
    key_compare comp;
};
```

Remarks

The comparison criterion provided by `value_compare` between `value_types` of whole elements contained by a map is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object `comp` of type `key_compare` stored in the function object provided by `value_compare` to compare the sort-key components of two elements.

For sets and multisets, which are simple containers where the key values are identical to the element values, `value_compare` is equivalent to `key_compare`; for maps and multimaps they are not, as the value of the type `pair` elements is not identical to the value of the element's key.

Example

See example for [value_comp](#) for an example of how to declare and use `value_compare`.

Requirements

Header: <map>

Namespace: std

See also

[binary_function Struct](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<memory>

11/9/2018 • 3 minutes to read • [Edit Online](#)

Defines a class, an operator, and several templates that help allocate and free objects.

Syntax

```
#include <memory>
```

Members

Functions

FUNCTION	DESCRIPTION
addressof	Gets the true address of an object.
align	Returns a pointer to a range of a given size, based on the provided alignment and starting address.
allocate_shared	Creates a <code>shared_ptr</code> to objects that are allocated and constructed for a given type with a specified allocator.
const_pointer_cast	Const cast to <code>shared_ptr</code> .
declare_no_pointers	Informs a garbage collector that the characters starting at a specified address and falling within the indicated block size contain no traceable pointers.
declare_reachable	Informs garbage collection that the indicated address is to allocated storage and is reachable.
default_delete	Deletes objects allocated with <code>operator new</code> . Suitable for use with <code>unique_ptr</code> .
dynamic_pointer_cast	Dynamic cast to <code>shared_ptr</code> .
get_deleter	Get deleter from <code>shared_ptr</code> .
get_pointer_safety	Returns the type of pointer safety assumed by any garbage collector.
get_temporary_buffer	Allocates temporary storage for a sequence of elements that does not exceed a specified number of elements.
make_shared	Creates and returns a <code>shared_ptr</code> that points to the allocated object constructed from zero or more arguments using the default allocator.

FUNCTION	DESCRIPTION
<code>make_unique</code>	Creates and returns a <code>unique_ptr</code> that points to the allocated object constructed from zero or more arguments.
<code>owner_less</code>	Allows ownership-based mixed comparisons of shared and weak pointers.
<code>pointer_safety</code>	An enumeration of all the possible return values for <code>get_pointer_safety</code> .
<code>return_temporary_buffer</code>	Deallocates the temporary memory that was allocated using the <code>get_temporary_buffer</code> template function.
<code>static_pointer_cast</code>	Static cast to <code>shared_ptr</code> .
<code>swap</code>	Swap two <code>shared_ptr</code> or <code>weak_ptr</code> objects.
<code>undecare_no_pointers</code>	Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size may now contain traceable pointers.
<code>undecare_reachable</code>	Informs a <code>garbage_collector</code> that a specified memory location is not reachable.
<code>uninitialized_copy</code>	Copies objects from a specified input range into an uninitialized destination range.
<code>uninitialized_copy_n</code>	Creates a copy of a specified number of elements from an input iterator. The copies are put in a forward iterator.
<code>uninitialized_fill</code>	Copies objects of a specified value into an uninitialized destination range.
<code>uninitialized_fill_n</code>	Copies objects of a specified value into specified number of elements an uninitialized destination range.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests for inequality between allocator objects of a specified class.
<code>operator==</code>	Tests for equality between allocator objects of a specified class.
<code>operator>=</code>	Tests for one allocator object being greater than or equal to a second allocator object, of a specified class.
<code>operator<</code>	Tests for one object being less than a second object of a specified class.
<code>operator<=</code>	Tests for one object being less than or equal to a second object of a specified class.

OPERATOR	DESCRIPTION
<code>operator></code>	Tests for one object being greater than a second object of a specified class.
<code>operator<<</code>	<code>shared_ptr</code> inserter.

Classes

CLASS	DESCRIPTION
<code>allocator</code>	The template class describes an object that manages storage allocation and freeing for arrays of objects of type Type .
<code>allocator_traits</code>	Describes an object that determines all the information that is needed by an allocator-enabled container.
<code>auto_ptr</code>	The template class describes an object that stores a pointer to an allocated object of type Type * that ensures the object to which it points gets deleted when its enclosing <code>auto_ptr</code> gets destroyed.
<code>bad_weak_ptr</code>	Reports bad <code>weak_ptr</code> exception.
<code>enabled_shared_from_this</code>	Helps generate a <code>shared_ptr</code> .
<code>pointer_traits</code>	Supplies information that is needed by an object of template class <code>allocator_traits</code> to describe an allocator with pointer type <code>Ptr</code> .
<code>raw_storage_iterator</code>	An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.
<code>shared_ptr</code>	Wraps a reference-counted smart pointer around a dynamically allocated object.
<code>unique_ptr</code>	Stores a pointer to an owned object. The pointer is owned by no other <code>unique_ptr</code> . The <code>unique_ptr</code> is destroyed when the owner is destroyed.
<code>weak_ptr</code>	Wraps a weakly linked pointer.

Specializations

<code>allocator<void></code>	A specialization of the template class <code>allocator</code> to type <code>void</code> , defining the only the member types that make sense in this specialized context.
------------------------------------	---

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<memory> functions

2/6/2019 • 20 minutes to read • [Edit Online](#)

addressof	align	allocate_shared
const_pointer_cast	declare_no_pointers	declare_reachable
default_delete	dynamic_pointer_cast	get_deleter
get_pointer_safety	get_temporary_buffer	make_shared
make_unique	owner_less	return_temporary_buffer
static_pointer_cast	swap (C++ Standard Library)	undecare_no_pointers
undecare_reachable	uninitialized_copy	uninitialized_copy_n
uninitialized_fill	uninitialized_fill_n	

addressof

Gets the true address of an object.

```
template <class T>
T* addressof(T& Val);
```

Parameters

Val

The object or function for which to obtain the true address.

Return Value

The actual address of the object or function referenced by *Val*, even if an overloaded `operator&()` exists.

Remarks

align

Fits storage of the given size—aligned by the given alignment specification—into the first possible address of the given storage.

```
void* align(
    size_t Alignment, // input
    size_t Size,      // input
    void*& Ptr,        // input/output
    size_t& Space      // input/output
);
```

Parameters

Alignment

The alignment bound to attempt.

Size

The size in bytes for the aligned storage.

Ptr

The starting address of the available contiguous storage pool to use. This parameter is also an output parameter, and is set to contain the new starting address if the alignment is successful. If `align()` is unsuccessful, this parameter is not modified.

Space

The total space available to `align()` to use in creating the aligned storage. This parameter is also an output parameter, and contains the adjusted space left in the storage buffer after the aligned storage and any associated overhead is subtracted.

If `align()` is unsuccessful, this parameter is not modified.

Return Value

A null pointer if the requested aligned buffer would not fit into the available space; otherwise, the new value of *Ptr*.

Remarks

The modified *Ptr* and *Space* parameters enable you to call `align()` repeatedly on the same buffer, possibly with different values for *Alignment* and *Size*. The following code snippet shows one use of `align()`.

```
#include <type_traits> // std::alignment_of()
#include <memory>
//...
char buffer[256]; // for simplicity
size_t alignment = std::alignment_of<int>::value;
void * ptr = buffer;
std::size_t space = sizeof(buffer); // Be sure this results in the true size of your buffer

while (std::align(alignment, sizeof(MyObj), ptr, space)) {
    // You now have storage the size of MyObj, starting at ptr, aligned on
    // int boundary. Use it here if you like, or save off the starting address
    // contained in ptr for later use.
    // ...
    // Last, move starting pointer and decrease available space before
    // the while loop restarts.
    ptr = reinterpret_cast<char*>(ptr) + sizeof(MyObj);
    space -= sizeof(MyObj);
}
// At this point, align() has returned a null pointer, signaling it is not
// possible to allow more aligned storage in this buffer.
```

allocate_shared

Creates a `shared_ptr` to objects that are allocated and constructed for a given type by using a specified allocator. Returns the `shared_ptr`.

```
template <class Type, class Allocator, class... Types>
shared_ptr<Type>
allocate_shared(Allocator Alloc, Types&&... Args);
```

Parameters

Alloc

The allocator used to create objects.

Args

The zero or more arguments that become the objects.

Remarks

The function creates the object `shared_ptr<Type>`, a pointer to `Type(Args...)` as allocated and constructed by *Alloc*.

const_pointer_cast

Const cast to shared_ptr.

```
template <class Ty, class Other>
shared_ptr<Ty>
const_pointer_cast(const shared_ptr<Other>& sp);
```

Parameters

Ty

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

Other

The argument shared pointer.

Remarks

The template function returns an empty shared_ptr object if `const_cast<Ty*>(sp.get())` returns a null pointer; otherwise it returns a [shared_ptr Class](#)<Ty> object that owns the resource that is owned by `sp`. The expression

`const_cast<Ty*>(sp.get())` must be valid.

Example

```
// std__memory__const_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int);
    std::shared_ptr<const int> sp1 =
        std::const_pointer_cast<const int>(sp0);

    *sp0 = 3;
    std::cout << "sp1 == " << *sp1 << std::endl;

    return (0);
}
```

```
sp1 == 3
```

declare_no_pointers

Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size contains no traceable pointers.

```
void declare_no_pointers(
    char* ptr,
    size_t _Size);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	Address of first character that no longer contains traceable pointers.
<i>_Size</i>	Size of block that starts at <i>ptr</i> that contains no traceable pointers.

Remarks

The function informs any garbage collector that the range of addresses [*ptr*, *ptr* + *_Size*) no longer contain traceable pointers. (Any pointers to allocated storage must not be dereferenced unless made reachable.)

declare_reachable

Informs garbage collection that the indicated address is to allocated storage and is reachable.

```
void declare_reachable(void* ptr);
```

Parameters

ptr

A pointer to a reachable, allocated, valid storage area.

Remarks

If *ptr* is not null, the function informs any garbage collector that *ptr* is hereafter reachable (points to valid allocated storage).

default_delete

Deletes objects allocated with **operator new**. Suitable for use with `unique_ptr`.

```
struct default_delete {
    constexpr default_delete() noexcept = default;
    template <class Other, class = typename enable_if<is_convertible<Other*, T*>::value, void>::type>>
    default_delete(const default_delete<Other>&) noexcept;
    void operator()(T* Ptr) const noexcept;
};
```

Parameters

Ptr

Pointer to the object to delete.

Other

The type of elements in the array to be deleted.

Remarks

The template class describes a `deleter` that deletes scalar objects allocated with **operator new**, suitable for use with template class `unique_ptr`. It also has the explicit specialization `default_delete<Type[]>`.

dynamic_pointer_cast

Dynamic cast to shared_ptr.

```
template <class Ty, class Other>
shared_ptr<Ty>
dynamic_pointer_cast(const shared_ptr<Other>& sp);
```

Parameters

Ty

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

sp

The argument shared pointer.

Remarks

The template function returns an empty shared_ptr object if `dynamic_cast<Ty*>(sp.get())` returns a null pointer; otherwise it returns a [shared_ptr Class](#)<Ty> object that owns the resource that is owned by *sp*. The expression

`dynamic_cast<Ty*>(sp.get())` must be valid.

Example

```
// std__memory__dynamic_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct base
{
    virtual ~base() {}
    int val;
};

struct derived
    : public base
{
};

int main()
{
    std::shared_ptr<base> sp0(new derived);
    std::shared_ptr<derived> sp1 =
        std::dynamic_pointer_cast<derived>(sp0);

    sp0->val = 3;
    std::cout << "sp1->val == " << sp1->val << std::endl;

    return (0);
}
```

```
sp1->val == 3
```

get_deleter

Get deleter from shared_ptr.

```
template <class D, class Ty>
D* get_deleter(const shared_ptr<Ty>& sp);
```

Parameters

D

The type of the deleter.

Ty

The type controlled by the shared pointer.

sp

The shared pointer.

Remarks

The template function returns a pointer to the deleter of type *D* that belongs to the [shared_ptr Class](#) object *sp*. If *sp* has no deleter or if its deleter is not of type *D* the function returns 0.

Example

```
// std__memory__get_deleter.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct base
{
    int val;
};

struct deleter
{
    void operator()(base *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<base> sp0(new base);

    sp0->val = 3;
    std::cout << "get_deleter(sp0) != 0 == " << std::boolalpha
        << (std::get_deleter<deleter>(sp0) != 0) << std::endl;

    std::shared_ptr<base> sp1(new base, deleter());

    sp0->val = 3;
    std::cout << "get_deleter(sp1) != 0 == " << std::boolalpha
        << (std::get_deleter<deleter>(sp1) != 0) << std::endl;

    return (0);
}
```

```
get_deleter(sp0) != 0 == false
get_deleter(sp1) != 0 == true
```

get_pointer_safety

Returns the type of pointer safety assumed by any garbage collector.

```
pointer_safety get_pointer_safety();
```

Remarks

The function returns the type of pointer safety assumed by any automatic garbage collector.

get_temporary_buffer

Allocates temporary storage for a sequence of elements that does not exceed a specified number of elements.

```
template <class Type>
pair<Type *, ptrdiff_t> get_temporary_buffer(ptrdiff_t count);
```

Parameters

count

The maximum number of elements requested for which memory is to be allocated.

Return Value

A `pair` whose first component is a pointer to the memory that was allocated, and whose second component gives the size of the buffer, indicating the largest number of elements that it could store.

Remarks

The function makes a request for memory and it may not succeed. If no buffer is allocated, then the function returns a pair, with the second component equal to zero and the first component equal to the null pointer.

This function should only be used for memory that is temporary.

Example

```
// memory_get_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

int main( )
{
    // Create an array of ints
    int intArray [ ] = { 10, 20, 30, 40, 100, 200, 300, 1000, 2000 };
    int count = sizeof ( intArray ) / sizeof ( int );
    cout << "The number of integers in the array is: "
         << count << "." << endl;

    pair<int *, ptrdiff_t> resultPair;
    resultPair = get_temporary_buffer<int>( count );

    cout << "The number of elements that the allocated memory\n"
         << "could store is given by: resultPair.second = "
         << resultPair.second << "." << endl;
}
```

```
The number of integers in the array is: 9.
The number of elements that the allocated memory
could store is given by: resultPair.second = 9.
```

make_shared

Creates and returns a `shared_ptr` that points to the allocated objects that are constructed from zero or more arguments by using the default allocator. Allocates and constructs both an object of the specified type and a `shared_ptr` to manage shared ownership of the object, and returns the `shared_ptr`.

```
template <class Type, class... Types>
shared_ptr<Type>
make_shared(Types&&... _Args);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Args</code>	Zero or more constructor arguments. The function infers which constructor overload to invoke based on the arguments that are provided.

Remarks

Use `make_shared` as a simple and more efficient way to create an object and a `shared_ptr` to manage shared access to the object at the same time. Semantically, these two statements are equivalent:

```
auto sp = std::shared_ptr<Example>(new Example(argument));
auto msp = std::make_shared<Example>(argument);
```

However, the first statement makes two allocations, and if the allocation of the `shared_ptr` fails after the allocation of the `Example` object has succeeded, then the unnamed `Example` object is leaked. The statement that uses `make_shared` is simpler because there's only one function call involved. It's more efficient because the library can make a single allocation for both the object and the smart pointer. This is both faster and leads to less memory fragmentation, and there is no chance of an exception on one allocation but not the other. Performance is improved by better locality for code that references the object and updates the reference counts in the smart pointer.

Consider using [make_unique](#) if you do not need shared access to the object. Use [allocate_shared](#) if you need to specify a custom allocator for the object. You can't use `make_shared` if your object requires a custom deleter, because there is no way to pass the deleter as an argument.

The following example shows how to create shared pointers to a type by invoking specific constructor overloads.

Example

```

// stl_make_shared.cpp
// Compile by using: cl /W4 /EHsc stl_make_shared.cpp
#include <iostream>
#include <string>
#include <memory>
#include <vector>

class Song {
public:
    std::wstring title_;
    std::wstring artist_;

    Song(std::wstring title, std::wstring artist) : title_(title), artist_(artist) {}
    Song(std::wstring title) : title_(title), artist_(L"Unknown") {}
};

void CreateSharedPointers() {
    // Okay, but less efficient to have separate allocations for
    // Song object and shared_ptr control block.
    auto song = new Song(L"Ode to Joy", L"Beethoven");
    std::shared_ptr<Song> sp0(song);

    // Use make_shared function when possible. Memory for control block
    // and Song object are allocated in the same call:
    auto sp1 = std::make_shared<Song>(L"Yesterday", L"The Beatles");
    auto sp2 = std::make_shared<Song>(L"Blackbird", L"The Beatles");

    // make_shared infers which constructor to use based on the arguments.
    auto sp3 = std::make_shared<Song>(L"Greensleeves");

    // The playlist vector makes copies of the shared_ptr pointers.
    std::vector<std::shared_ptr<Song>> playlist;
    playlist.push_back(sp0);
    playlist.push_back(sp1);
    playlist.push_back(sp2);
    playlist.push_back(sp3);
    playlist.push_back(sp1);
    playlist.push_back(sp2);
    for (auto&& sp : playlist) {
        std::wcout << L"Playing " << sp->title_ <<
            L" by " << sp->artist_ << L", use count: " <<
            sp.use_count() << std::endl;
    }
}

int main() {
    CreateSharedPointers();
}

```

The example produces this output:

```

Playing Ode to Joy by Beethoven, use count: 2
Playing Yesterday by The Beatles, use count: 3
Playing Blackbird by The Beatles, use count: 3
Playing Greensleeves by Unknown, use count: 2
Playing Yesterday by The Beatles, use count: 3
Playing Blackbird by The Beatles, use count: 3

```

make_unique

Creates and returns a [unique_ptr](#) to an object of the specified type, which is constructed by using the specified arguments.


```

// make_unique<T>
template <class T, class... Types>
unique_ptr<T>
make_unique(Types&&... Args)
{
    return (unique_ptr<T>(new T(forward<Types>(Args)...)));
}

// make_unique<T[]>
template <class T>
make_unique(size_t Size)
{
    return (unique_ptr<T>(new Elem[Size]()));
}

// make_unique<T[N]> disallowed
template <class T, class... Types>
typename enable_if<extent<T>::value != 0, void>::type
make_unique(Types&&...) = delete;

```

Parameters

T

The type of the object that the `unique_ptr` will point to.

Types

The types of the constructor arguments specified by *Args*.

Args

The arguments to be passed to the constructor of the object of type *T*.

Elem

An array of elements of type *T*.

Size

The number of elements to allocate space for in the new array.

Remarks

The first overload is used for single objects, the second overload is invoked for arrays, and the third overload prevents you from specifying an array size in the type argument (`make_unique<T[N]>`); this construction is not supported by the current standard. When you use `make_unique` to create a `unique_ptr` to an array, you have to initialize the array elements separately. If you are considering this overload, perhaps a better choice is to use a [std::vector](#).

Because `make_unique` is carefully implemented for exception safety, we recommend that you use `make_unique` instead of directly calling `unique_ptr` constructors.

Example

The following example shows how to use `make_unique`. For more examples, see [How to: Create and Use unique_ptr Instances](#).

```

class Animal
{
private:
    std::wstring genus;
    std::wstring species;
    int age;
    double weight;
public:
    Animal(const wstring&, const wstring&, int, double){/*...*/ }
    Animal(){}
};

void MakeAnimals()
{
    // Use the Animal default constructor.
    unique_ptr<Animal> p1 = make_unique<Animal>();

    // Use the constructor that matches these arguments
    auto p2 = make_unique<Animal>(L"Felis", L"Catus", 12, 16.5);

    // Create a unique_ptr to an array of 5 Animals
    unique_ptr<Animal[]> p3 = make_unique<Animal[]>(5);

    // Initialize the elements
    p3[0] = Animal(L"Rattus", L"norvegicus", 3, 2.1);
    p3[1] = Animal(L"Corynorhinus", L"townsendii", 4, 1.08);

    // auto p4 = p2; //C2280

    vector<unique_ptr<Animal>> vec;
    // vec.push_back(p2); //C2280
    // vector<unique_ptr<Animal>> vec2 = vec; // C2280

    // OK. p2 no longer points to anything
    vec.push_back(std::move(p2));

    // unique_ptr overloads operator bool
    wcout << boolalpha << (p2 == false) << endl; // Prints "true"

    // OK but now you have two pointers to the same memory location
    Animal* pAnimal = p2.get();

    // OK. p2 no longer points to anything
    Animal* p5 = p2.release();
}

```

When you see error C2280 in connection with a `unique_ptr`, it is almost certainly because you are attempting to invoke its copy constructor, which is a deleted function.

owner_less

Allows ownership-based mixed comparisons of shared and weak pointers. Returns **true** if the left parameter is ordered before right parameter by the member function `owner_before`.

```

template <class Type>
struct owner_less; // not defined

template <class Type>
struct owner_less<shared_ptr<Type>> {
    bool operator()(
        const shared_ptr<Type>& left,
        const shared_ptr<Type>& right);

    bool operator()(
        const shared_ptr<Type>& left,
        const weak_ptr<Type>& right);

    bool operator()(
        const weak_ptr<Type>& left,
        const shared_ptr<Type>& right);
};

template <class Type>
struct owner_less<weak_ptr<Type>>
    bool operator()(
        const weak_ptr<Type>& left,
        const weak_ptr<Type>& right);

    bool operator()(
        const weak_ptr<Type>& left,
        const shared_ptr<Type>& right);

    bool operator()(
        const shared_ptr<Type>& left,
        const weak_ptr<Type>& right);
};

```

Parameters

_left

A shared or weak pointer.

right

A shared or weak pointer.

Remarks

The template classes define all their member operators as returning `left.owner_before(right)`.

return_temporary_buffer

Deallocates the temporary memory that was allocated using the `get_temporary_buffer` template function.

```

template <class Type>
void return_temporary_buffer(Type* _Pbuf);

```

Parameters

_Pbuf

A pointer to the memory to be deallocated.

Remarks

This function should only be used for memory that is temporary.

Example

```

// memory_ret_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

int main( )
{
    // Create an array of ints
    int intArray [ ] = { 10, 20, 30, 40, 100, 200, 300 };
    int count = sizeof ( intArray ) / sizeof ( int );
    cout << "The number of integers in the array is: "
         << count << "." << endl;

    pair<int *, ptrdiff_t> resultPair;
    resultPair = get_temporary_buffer<int>( count );

    cout << "The number of elements that the allocated memory\n"
         << " could store is given by: resultPair.second = "
         << resultPair.second << "." << endl;

    int* tempBuffer = resultPair.first;

    // Deallocates memory allocated with get_temporary_buffer
    return_temporary_buffer ( tempBuffer );
}

```

```

The number of integers in the array is: 7.
The number of elements that the allocated memory
could store is given by: resultPair.second = 7.

```

static_pointer_cast

Static cast to shared_ptr.

```

template <class Ty, class Other>
shared_ptr<Ty>
static_pointer_cast(const shared_ptr<Other>& sp);

```

Parameters

Ty

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

Other

The argument shared pointer.

Remarks

The template function returns an empty shared_ptr object if `sp` is an empty `shared_ptr` object; otherwise it returns a `shared_ptr Class<Ty>` object that owns the resource that is owned by `sp`. The expression

```
static_cast<Ty*>(sp.get())
```

must be valid.

Example

```

// std__memory__static_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct base
{
    int val;
};

struct derived
    : public base
{
};

int main()
{
    std::shared_ptr<base> sp0(new derived);
    std::shared_ptr<derived> sp1 =
        std::static_pointer_cast<derived>(sp0);

    sp0->val = 3;
    std::cout << "sp1->val == " << sp1->val << std::endl;

    return (0);
}

```

```
sp1->val == 3
```

swap (C++ Standard Library)

Swap two `shared_ptr` or `weak_ptr` objects.

```

template <class Ty, class Other>
void swap(shared_ptr<Ty>& left, shared_ptr<Other>& right);

template <class Ty, class Other>
void swap(weak_ptr<Ty>& left, weak_ptr<Other>& right);

```

Parameters

Ty

The type controlled by the left shared/weak pointer.

Other

The type controlled by the right shared/weak pointer.

left

The left shared/weak pointer.

right

The right shared/weak pointer.

Remarks

The template functions call `left.swap(right)`.

Example

```

// std_memory_swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}

```

```

*sp1 == 5
*sp1 == 10
*sp1 == 5

*wp1 == 5
*wp1 == 10
*wp1 == 5

```

undeclare_no_pointers

Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size may now contain traceable pointers.

```

void undeclare_no_pointers(
    char* ptr,
    size_t _Size);

```

Remarks

The function informs any garbage collector that the range of addresses `[ptr, ptr + _Size)` may now contain traceable pointers.

undeclare_reachable

Revokes a declaration of reachability for a specified memory location.

```
template <class Type>
Type *undeclare_reachable(Type* ptr);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the memory address to be declared not reachable.

Remarks

If *ptr* is not **nullptr**, the function informs any garbage collector that *ptr* is no longer reachable. It returns a safely-derived pointer that compares equal to *ptr*.

uninitialized_copy

Copies objects from a specified source range into an uninitialized destination range.

```
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator dest);
```

Parameters

first

An input iterator addressing the first element in the source range.

last

An input iterator addressing the last element in the source range.

dest

A forward iterator addressing the first element in the destination range.

Return Value

A forward iterator addressing the first position beyond the destination range, unless the source range was empty.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
while (first != last) {
    new (static_cast<void*>(&* dest++))
        typename iterator_traits<InputIterator>::value_type(*first++);
}
return dest;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Example

```

// memory_uninit_copy.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

class Integer
{
public:
    Integer(int x) : val(x) {}
    int get() { return val; }
private:
    int val;
};

int main()
{
    int Array[] = { 10, 20, 30, 40 };
    const int N = sizeof(Array) / sizeof(int);

    int i;
    cout << "The initialized Array contains " << N << " elements: ";
    for (i = 0; i < N; i++)
    {
        cout << " " << Array[i];
    }
    cout << endl;

    Integer* ArrayPtr = (Integer*)malloc(N * sizeof(int));
    Integer* LArrayPtr = uninitialized_copy(
        Array, Array + N, ArrayPtr); // C4996

    cout << "Address of position after the last element in the array is: "
        << &Array[0] + N << endl;
    cout << "The iterator returned by uninitialized_copy addresses: "
        << (void*)LArrayPtr << endl;
    cout << "The address just beyond the last copied element is: "
        << (void*)(ArrayPtr + N) << endl;

    if ((&Array[0] + N) == (void*)LArrayPtr)
        cout << "The return value is an iterator "
            << "pointing just beyond the original array." << endl;
    else
        cout << "The return value is an iterator "
            << "not pointing just beyond the original array." << endl;

    if ((void*)LArrayPtr == (void*)(ArrayPtr + N))
        cout << "The return value is an iterator "
            << "pointing just beyond the copied array." << endl;
    else
        cout << "The return value is an iterator "
            << "not pointing just beyond the copied array." << endl;

    free(ArrayPtr);

    cout << "Note that the exact addresses returned will vary\n"
        << "with the memory allocation in individual computers."
        << endl;
}

```

uninitialized_copy_n

Creates a copy of a specified number of elements from an input iterator. The copies are put in a forward iterator.


```
template <class InputIterator, class Size, class ForwardIterator>
ForwardIterator uninitialized_copy_n(
    InputIterator first,
    Size count,
    ForwardIterator dest);
```

Parameters

first

An input iterator that refers to the object to copy.

count

A signed or unsigned integer type specifying the number of times to copy the object.

dest

A forward iterator that refers to where the new copies go.

Return Value

A forward iterator that addresses the first position beyond the destination. If the source range was empty, the iterator addresses *first*.

Remarks

The template function effectively executes the following:

```
for (; 0 < count; --count)
    new (static_cast<void*>(&* dest++))
        typename iterator_traits<InputIterator>::value_type(*first++);
return dest;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill

Copies objects of a specified value into an uninitialized destination range.

```
template <class ForwardIterator, class Type>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const Type& val);
```

Parameters

first

A forward iterator addressing the first element in the destination range that is to be initiated.

last

A forward iterator addressing the last element in the destination range that is to be initiated.

val

The value to be used to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```

while (first != last)
    new (static_cast<void*>(&* first ++))
        typename iterator_traits<ForwardIterator>::value_type (val);

```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Example

```

// memory_uninit_fill.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

class Integer {          // No default constructor
public:
    Integer( int x ) : val( x ) {}
    int get( ) { return val; }
private:
    int val;
};

int main( )
{
    const int N = 10;
    Integer val ( 25 );
    Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
    uninitialized_fill( Array, Array + N, val );
    int i;
    cout << "The initialized Array contains: ";
    for ( i = 0 ; i < N; i++ )
    {
        cout << Array [ i ].get( ) << " ";
    }
    cout << endl;
}

```

The initialized Array contains: 25 25 25 25 25 25 25 25 25 25

uninitialized_fill_n

Copies objects of a specified value into specified number of elements into an uninitialized destination range.

```

template <class FwdIt, class Size, class Type>
void uninitialized_fill_n(ForwardIterator first, Size count, const Type& val);

```

Parameters

first

A forward iterator addressing the first element in the destination range to be initiated.

count

The number of elements to be initialized.

val

The value to be used to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
while (0 < count--)
    new (static_cast<void*>(&* first++))
        typename iterator_traits<ForwardIterator>::value_type(val);
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Example

```
// memory_uninit_fill_n.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

class Integer {    // No default constructor
public:
    Integer( int x ) : val( x ) {}
    int get( ) { return val; }
private:
    int val;
};

int main() {
    const int N = 10;
    Integer val ( 60 );
    Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
    uninitialized_fill_n( Array, N, val ); // C4996
    int i;
    cout << "The uninitialized Array contains: ";
    for ( i = 0 ; i < N; i++ )
        cout << Array [ i ].get( ) << " ";
}
```

See also

[`<memory>`](#)

<memory> operators

11/9/2018 • 4 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<<</code>	<code>operator<=</code>
<code>operator==</code>		

operator!=

Tests for inequality between objects.

```
template <class Type, class Other>
bool operator!=(
    const allocator<Type>& left,
    const allocator<Other>& right) throw();

template <class T, class Del1, class U, class Del2>
bool operator!=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U&, Del2>& right);

template <class Ty1, class Ty2>
bool operator!=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left

One of the objects to be tested for inequality.

right

One of the objects to be tested for inequality.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

Return Value

true if the objects are not equal; **false** if objects are equal.

Remarks

The first template operator returns false. (All default allocators are equal.)

The second and third template operators return `!(left == right)`.

Example

```

// memory_op_me.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    allocator<double> Alloc;
    vector <char>:: allocator_type v1Alloc;

    if ( Alloc != v1Alloc )
        cout << "The allocator objects Alloc & v1Alloc not are equal."
              << endl;
    else
        cout << "The allocator objects Alloc & v1Alloc are equal."
              << endl;
}

```

The allocator objects Alloc & v1Alloc are equal.

Example

```

// std__memory__operator_ne.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(0));
    std::shared_ptr<int> sp1(new int(0));

    std::cout << "sp0 != sp0 == " << std::boolalpha
              << (sp0 != sp0) << std::endl;
    std::cout << "sp0 != sp1 == " << std::boolalpha
              << (sp0 != sp1) << std::endl;

    return (0);
}

```

```

sp0 != sp0 == false
sp0 != sp1 == true

```

operator==

Tests for equality between objects.

```

template <class Type, class Other>
bool operator==(
    const allocator<Type>& left,
    const allocator<Other>& right) throw();

template <class Ty1, class Del1, class Ty2, class Del2>
bool operator==(
    const unique_ptr<Ty1, Del1>& left,
    const unique_ptr<Ty2, Del2>& right);

template <class Ty1, class Ty2>
bool operator==(
    const shared_ptr<Ty1>& left;,
    const shared_ptr<Ty2>& right);

```

Parameters

left

One of the objects to be tested for equality.

right

One of the objects to be tested for equality.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

Return Value

true if the objects are equal, **false** if objects are not equal.

Remarks

The first template operator returns true. (All default allocators are equal.)

The second and third template operators return `left.get() == right.get()`.

Example

```

// memory_op_eq.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    allocator<char> Alloc;
    vector<int>:: allocator_type v1Alloc;

    allocator<char> cAlloc(Alloc);
    allocator<int> cv1Alloc(v1Alloc);

    if ( cv1Alloc == v1Alloc )
        cout << "The allocator objects cv1Alloc & v1Alloc are equal."
              << endl;
    else
        cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
              << endl;

    if ( cAlloc == Alloc )
        cout << "The allocator objects cAlloc & Alloc are equal."
              << endl;
    else
        cout << "The allocator objects cAlloc & Alloc are not equal."
              << endl;
}

```

The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.

Example

```

// std_memory_operator_eq.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(0));
    std::shared_ptr<int> sp1(new int(0));

    std::cout << "sp0 == sp0 == " << std::boolalpha
              << (sp0 == sp0) << std::endl;
    std::cout << "sp0 == sp1 == " << std::boolalpha
              << (sp0 == sp1) << std::endl;

    return (0);
}

```

sp0 == sp0 == true
sp0 == sp1 == false

operator>=

Tests for one object being greater than or equal to a second object.

```

template <class T, class Del1, class U, class Del2>
bool operator>=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U, Del2>& right);

template <class Ty1, class Ty2>
bool operator>=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);

```

Parameters

left

One of the objects to be compared.

right

One of the objects to be compared.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

Remarks

The template operators return `left.get() >= right.get()` .

operator<

Tests for one object being less than a second object.

```

template <class T, class Del1, class U, class Del2>
bool operator<(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U, Del2>& right);

template <class Ty1, class Ty2>
bool operator<(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);

```

Parameters

left

One of the objects to be compared.

right

One of the objects to be compared.

Ty1

The type controlled by the left pointer.

Ty2

The type controlled by the right pointer.

operator<=

Tests for one object being less than or equal to a second object.


```
template <class T, class Del1, class U, class Del2>
bool operator<=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U&, Del2>& right);

template <class Ty1, class Ty2>
bool operator<=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left

One of the objects to be compared.

right

One of the objects to be compared.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

Remarks

The template operators return `left.get() <= right.get()`

operator>

Tests for one object being greater than a second object.

```
template <class Ty1, class Del1, class Ty2, class Del2>
bool operator>(
    const unique_ptr<Ty1, Del1>& left,
    const unique_ptr<Ty2&, Del2>& right);

template <class Ty1, class Ty2>
bool operator>(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left

One of the objects to be compared.

right

One of the objects to be compared.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

operator<<

Writes the shared pointer to the stream.

```
template <class Elem, class Tr, class Ty>
std::basic_ostream<Elem, Tr>& operator<<(std::basic_ostream<Elem, Tr>& out,
    shared_ptr<Ty>& sp);
```

Parameters

Elem

The type of the stream element.

Tr

The type the stream element traits.

Ty

The type controlled by the shared pointer.

out

The output stream.

sp

The shared pointer.

Remarks

The template function returns `out << sp.get()` .

Example

```
// std__memory__operator_sl.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));

    std::cout << "sp0 == " << sp0 << " (varies)" << std::endl;

    return (0);
}
```

```
sp0 == 3f3040 (varies)
```

See also

[<memory>](#)

<memory> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

`pointer_safety`

pointer_safety Enumeration

The enumeration of possible values returned by `get_pointer_safety`.

```
class pointer_safety { relaxed, preferred, strict };
```

Remarks

The scoped **enum** defines the values that can be returned by `get_pointer_safety()`:

`relaxed` -- pointers not safely derived (obviously pointers to declared or allocated objects) are treated the same as those safely derived.

`preferred` -- as before, but pointers not safely derived should not be dereferenced.

`strict` -- pointers not safely derived might be treated differently than those safely derived.

See also

[<memory>](#)

allocator Class

11/9/2018 • 18 minutes to read • [Edit Online](#)

The template class describes an object that manages storage allocation and freeing for arrays of objects of type `Type`. An object of class `allocator` is the default allocator object specified in the constructors for several container template classes in the C++ Standard Library.

Syntax

```
template <class Type>
class allocator
```

Parameters

Type

The type of object for which storage is being allocated or deallocated.

Remarks

All the C++ Standard Library containers have a template parameter that defaults to `allocator`. Constructing a container with a custom allocator provide control over allocation and freeing of that container's elements.

For example, an allocator object might allocate storage on a private heap or in shared memory, or it might optimize for small or large object sizes. It might also specify, through the type definitions it supplies, that elements be accessed through special accessor objects that manage shared memory, or perform automatic garbage collection. Hence, a class that allocates storage using an allocator object should use these types for declaring pointer and reference objects, as the containers in the C++ Standard Library do.

(C++98/03 only) When you derive from allocator class, you have to provide a `rebind` struct, whose `_Other` typedef references your newly-derived class.

Thus, an allocator defines the following types:

- `pointer` behaves like a pointer to `Type`.
- `const_pointer` behaves like a const pointer to `Type`.
- `reference` behaves like a reference to `Type`.
- `const_reference` behaves like a const reference to `Type`.

These `Type`s specify the form that pointers and references must take for allocated elements. (`allocator::pointer` is not necessarily the same as `Type*` for all allocator objects, even though it has this obvious definition for class `allocator`.)

C++11 and later: To enable move operations in your allocator, use the minimal allocator interface and implement copy constructor, `==` and `!=` operators, `allocate` and `deallocate`. For more information and an example, see [Allocators](#)

Members

Constructors

CONSTRUCTOR	DESCRIPTION
allocator	Constructors used to create <code>allocator</code> objects.

Typedefs

TYPE NAME	DESCRIPTION
const_pointer	A type that provides a constant pointer to the type of object managed by the allocator.
const_reference	A type that provides a constant reference to type of object managed by the allocator.
difference_type	A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.
pointer	A type that provides a pointer to the type of object managed by the allocator.
reference	A type that provides a reference to the type of object managed by the allocator.
size_type	An unsigned integral type that can represent the length of any sequence that an object of template class <code>allocator</code> can allocate.
value_type	A type that is managed by the allocator.

Member functions

MEMBER FUNCTION	DESCRIPTION
address	Finds the address of an object whose value is specified.
allocate	Allocates a block of memory large enough to store at least some specified number of elements.
construct	Constructs a specific type of object at a specified address that is initialized with a specified value.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
destroy	Calls an objects destructor without deallocating the memory where the object was stored.
max_size	Returns the number of elements of type <code>Type</code> that could be allocated by an object of class <code>allocator</code> before the free memory is used up.
rebind	A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns one <code>allocator</code> object to another <code>allocator</code> object.

Requirements

Header: <memory>

Namespace: std

allocator::address

Finds the address of an object whose value is specified.

```
pointer address(reference val) const;
const_pointer address(const_reference val) const;
```

Parameters

val

The const or nonconst value of the object whose address is being searched for.

Return Value

A const or nonconst pointer to the object found of, respectively, const or nonconst value.

Remarks

The member functions return the address of *val*, in the form that pointers must take for allocated elements.

Example

```

// allocator_address.cpp
// compile with: /EHsc
#include <memory>
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <int> v1;
    vector <int>::iterator v1Iter;
    vector <int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::const_pointer v1Ptr;
    const int k = 8;
    v1Ptr = v1Alloc.address( *find(v1.begin( ), v1.end( ), k) );
    // v1Ptr = v1Alloc.address( k );
    cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}

```

```

The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 8.

```

allocator::allocate

Allocates a block of memory large enough to store at least some specified number of elements.

```

pointer allocate(size_type count, const void* _Hint);

```

Parameters

count

The number of elements for which sufficient storage is to be allocated.

_Hint

A const pointer that may assist the allocator object satisfy the request for storage by locating the address of an object allocated prior to the request.

Return Value

A pointer to the allocated object or null if memory was not allocated.

Remarks

The member function allocates storage for an array of count elements of type `Type`, by calling operator `new(count)`. It returns a pointer to the allocated object. The hint argument helps some allocators in improving

locality of reference; a valid choice is the address of an object earlier allocated by the same allocator object and not yet deallocated. To supply no hint, use a null pointer argument instead.

Example

```
// allocator_allocate.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    allocator<int> v1Alloc;
    allocator<int>::pointer v1aPtr;
    v1aPtr = v1Alloc.allocate ( 10 );

    int i;
    for ( i = 0 ; i < 10 ; i++ )
    {
        v1aPtr[ i ] = i;
    }

    for ( i = 0 ; i < 10 ; i++ )
    {
        cout << v1aPtr[ i ] << " ";
    }
    cout << endl;
    v1Alloc.deallocate( v1aPtr, 10 );
}
```

0 1 2 3 4 5 6 7 8 9

allocator::allocator

Constructors used to create allocator objects.

```
allocator();
allocator(const allocator<Type>& right);
template <class Other>
allocator(const allocator<Other>& right);
```

Parameters

right

The allocator object to be copied.

Remarks

The constructor does nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example


```

// allocator_allocator.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
    Int( int i )
    {
        cout << "Constructing " << ( void* )this << endl;
        x = i;
        bIsConstructed = true;
    };
    ~Int( )
    {
        cout << "Destructing " << ( void* )this << endl;
        bIsConstructed = false;
    };
    Int &operator++( )
    {
        x++;
        return *this;
    };
    int x;
private:
    bool bIsConstructed;
};

int main( )
{
    allocator<double> Alloc;
    vector <Int>:: allocator_type v1Alloc;

    allocator<double> cAlloc(Alloc);
    allocator<Int> cv1Alloc(v1Alloc);

    if ( cv1Alloc == v1Alloc )
        cout << "The allocator objects cv1Alloc & v1Alloc are equal."
              << endl;
    else
        cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
              << endl;

    if ( cAlloc == Alloc )
        cout << "The allocator objects cAlloc & Alloc are equal."
              << endl;
    else
        cout << "The allocator objects cAlloc & Alloc are not equal."
              << endl;
}

```

```

The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.

```

allocator::const_pointer

A type that provides a constant pointer to the type of object managed by the allocator.

```
typedef const value_type *const_pointer;
```

Remarks

The pointer type describes an object `ptr` that can designate, through the expression `*ptr`, any const object that an object of template class allocator can allocate.

Example

```
// allocator_const_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <int> v1;
    vector <int>::iterator v1Iter;
    vector <int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( i * 2 );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::const_pointer v1Ptr;
    const int k = 10;
    v1Ptr = v1Alloc.address( k );

    cout << "The integer's address found has a value of: "
        << *v1Ptr << "." << endl;
}
```

```
The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The integer's address found has a value of: 10.
```

allocator::const_reference

A type that provides a constant reference to type of object managed by the allocator.

```
typedef const value_type& const_reference;
```

Remarks

The reference type describes an object that can designate any const object that an object of template class allocator can allocate.

Example

```

// allocator_const_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <double> v;
    vector <double> ::iterator vIter, vfIter;
    vector <double> :: allocator_type vAlloc;

    int j;
    for ( j = 1 ; j <= 7 ; j++ )
    {
        v.push_back( 100.0 * j );
    }

    cout << "The original vector v is:\n ( " ;
    for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vfIter = v.begin( );
    allocator<double>::const_reference vcref =*vfIter;
    cout << "The value of the element referred to by vref is: "
        << vcref << ",\n the first element in the vector." << endl;

    // const references can have their elements modified,
    // so the following would generate an error:
    // vcref = 150;
    // but the value of the first element could be modified through
    // its nonconst iterator and the const reference would remain valid
    *vfIter = 175;
    cout << "The value of the element referred to by vcref,"
        << "\n after nofication through its nonconst iterator, is: "
        << vcref << "." << endl;
}

```

```

The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
the first element in the vector.
The value of the element referred to by vcref,
after nofication through its nonconst iterator, is: 175.

```

allocator::construct

Constructs a specific type of object at a specified address that is initialized with a specified value.

```

void construct(pointer ptr, const Type& val);
void construct(pointer ptr, Type&& val);
template <class _Other>
void construct(pointer ptr, _Other&&... val);

```

Parameters

ptr

A pointer to the location where the object is to be constructed.

val

The value with which the object being constructed is to be initialized.

Remarks

The first member function is equivalent to **new** ((`void *`) `ptr`) **Type** (`val`).

Example

```
// allocator_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
    vector <int> v1;
    vector <int>::iterator v1Iter;
    vector <int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( 3 * i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::pointer v1PtrA;
    int kA = 6, kB = 7;
    v1PtrA = v1Alloc.address( *find( v1.begin( ), v1.end( ), kA ) );
    v1Alloc.destroy ( v1PtrA );
    v1Alloc.construct ( v1PtrA , kB );

    cout << "The modified vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;
}
```

```
The original vector v1 is:
( 3 6 9 12 15 18 21 ).
The modified vector v1 is:
( 3 7 9 12 15 18 21 ).
```

allocator::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
void deallocate(pointer ptr, size_type count);
```

Parameters

ptr

A pointer to the first object to be deallocated from storage.

count

The number of objects to be deallocated from storage.

Remarks

The member function frees storage for the array of *count* objects of type `Type` beginning at *ptr*, by calling `operator delete(ptr)`. The pointer *ptr* must have been returned earlier by a call to [allocate](#) for an allocator object that compares equal to ***this**, allocating an array object of the same size and type. `deallocate` never throws an exception.

Example

For an example using the member function, see [allocator::allocate](#).

allocator::destroy

Calls an objects destructor without deallocating the memory where the object was stored.

```
void destroy(pointer ptr);
```

Parameters

ptr

A pointer designating the address of the object to be destroyed.

Remarks

The member function destroys the object designated by *ptr*, by calling the destructor `ptr->Type::~~Type`.

Example

```

// allocator_destroy.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
    vector<int> v1;
    vector<int>::iterator v1Iter;
    vector<int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::pointer v1PtrA;
    int kA = 12, kB = -99;
    v1PtrA = v1Alloc.address( *find(v1.begin( ), v1.end( ), kA) );
    v1Alloc.destroy ( v1PtrA );
    v1Alloc.construct ( v1PtrA , kB );

    cout << "The modified vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;
}

```

```

The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The modified vector v1 is:
( 2 4 6 8 10 -99 14 ).

```

allocator::difference_type

A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.

```
typedef ptrdiff_t difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of template class allocator can allocate.

Example

```

// allocator_diff_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector<int> v1;
    vector<int>::iterator v1Iter;
    vector<int>:: allocator_type v1Alloc;

    int i;
    for ( i = 0 ; i <= 7 ; i++ )
    {
        v1.push_back( i * 2 );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::const_pointer v1PtrA, v1PtrB;
    const int kA = 4, kB =12;
    v1PtrA = v1Alloc.address( kA );
    v1PtrB = v1Alloc.address( kB );
    allocator<int>::difference_type v1diff = *v1PtrB - *v1PtrA;

    cout << "Pointer v1PtrA addresses " << *v1PtrA << "." << endl;
    cout << "Pointer v1PtrB addresses " << *v1PtrB << "." << endl;
    cout << "The difference between the integer's addresses is: "
        << v1diff << "." << endl;
}

```

```

The original vector v1 is:
( 0 2 4 6 8 10 12 14 ).
Pointer v1PtrA addresses 4.
Pointer v1PtrB addresses 12.
The difference between the integer's addresses is: 8.

```

allocator::max_size

Returns the number of elements of type `Type` that could be allocated by an object of class `allocator` before the free memory is used up.

```

size_type max_size() const;

```

Return Value

The number of elements that could be allocated.

Example

```

// allocator_max_size.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector<int> v1;
    vector<int>::iterator v1Iter;
    vector<int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    vector<double> v2;
    vector<double> ::iterator v2Iter;
    vector<double> :: allocator_type v2Alloc;
    allocator<int>::size_type v1size;
    v1size = v1Alloc.max_size( );

    cout << "The number of integers that can be allocated before\n"
        << " the free memory in the vector v1 is used up is: "
        << v1size << "." << endl;

    int ii;
    for ( ii = 1 ; ii <= 7 ; ii++ )
    {
        v2.push_back( ii * 10.0 );
    }

    cout << "The original vector v2 is:\n ( " ;
    for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ; v2Iter++ )
        cout << *v2Iter << " ";
    cout << ")." << endl;
    allocator<double>::size_type v2size;
    v2size = v2Alloc.max_size( );

    cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v2 is used up is: "
        << v2size << "." << endl;
}

```

allocator::operator=

Assigns one allocator object to another allocator object.

```

template <class Other>
allocator<Type>& operator=(const allocator<Other>& right);

```

Parameters

right

An allocator object to be assigned to another such object.

Return Value

A reference to the allocator object

Remarks

The template assignment operator does nothing. In general, however, an allocator object assigned to another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example

```
// allocator_op_assign.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
    Int(int i)
    {
        cout << "Constructing " << ( void* )this << endl;
        x = i;
        bIsConstructed = true;
    };
    ~Int( ) {
        cout << "Destructing " << ( void* )this << endl;
        bIsConstructed = false;
    };
    Int &operator++( )
    {
        x++;
        return *this;
    };
    int x;
private:
    bool bIsConstructed;
};

int main( )
{
    allocator<Int> Alloc;
    allocator<Int> cAlloc ;
    cAlloc = Alloc;
}
```

allocator::pointer

A type that provides a pointer to the type of object managed by the allocator.

```
typedef value_type *pointer;
```

Remarks

The pointer type describes an object `ptr` that can designate, through the expression ***ptr**, any object that an object of template class allocator can allocate.

Example

```

// allocator_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <int> v1;
    vector <int>::iterator v1Iter;
    vector <int>:: allocator_type v1Alloc;

    int i;
    for ( i = 1 ; i <= 7 ; i++ )
    {
        v1.push_back( 3 * i );
    }

    cout << "The original vector v1 is:\n( " ;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    allocator<int>::const_pointer v1Ptr;
    const int k = 12;
    v1Ptr = v1Alloc.address( k );

    cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}

```

```

The original vector v1 is:
( 3 6 9 12 15 18 21 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 12.

```

allocator::rebind

A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

```

struct rebind {     typedef allocator<_Other> other ;     };

```

Parameters

other

The type of element for which memory is being allocated.

Remarks

This structure is useful for allocating memory for type that differs from the element type of the container being implemented.

The member template class defines the type *other*. Its sole purpose is to provide the type name **allocator<_Other>**, given the type name **allocator< Type>**.

For example, given an allocator object `a1` of type `A`, you can allocate an object of type `_Other` with the expression:

```
A::rebind<Other>::other(al).allocate(1, (Other *)0)
```

Or, you can name its pointer type by writing the type:

```
A::rebind<Other>::other::pointer
```

Example

```
// allocator_rebind.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

typedef vector<int>::allocator_type IntAlloc;
int main( )
{
    IntAlloc v1Iter;
    vector<int> v1;

    IntAlloc::rebind<char>::other::pointer pszC =
        IntAlloc::rebind<char>::other(v1.get_allocator()).allocate(1, (void *)0);

    int * pInt = v1Iter.allocate(10);
}
```

allocator::reference

A type that provides a reference to the type of object managed by the allocator.

```
typedef value_type& reference;
```

Remarks

The reference type describes an object that can designate any object that an object of template class allocator can allocate.

Example

```

// allocator_reference.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <double> v;
    vector <double> ::iterator vIter, vfIter;
    vector <double> :: allocator_type vAlloc;

    int j;
    for ( j = 1 ; j <= 7 ; j++ )
    {
        v.push_back( 100.0 * j );
    }

    cout << "The original vector v is:\n ( " ;
    for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vfIter = v.begin( );
    allocator<double>::reference vref =*vfIter;
    cout << "The value of the element referred to by vref is: "
        << vref << ",\n the first element in the vector." << endl;

    // nonconst references can have their elements modified
    vref = 150;
    cout << "The element referred to by vref after being modified is: "
        << vref << "." << endl;
}

```

```

The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
the first element in the vector.
The element referred to by vref after being modified is: 150.

```

allocator::size_type

An unsigned integral type that can represent the length of any sequence that an object of template class allocator can allocate.

```
typedef size_t size_type;
```

Example

```

// allocator_size_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
    vector <double> v;
    vector <double> ::iterator vIter;
    vector <double> :: allocator_type vAlloc;

    int j;
    for ( j = 1 ; j <= 7 ; j++ )
    {
        v.push_back( 100.0 * j );
    }

    cout << "The original vector v is:\n ( " ;
    for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    allocator<double>::size_type vsize;
    vsize = vAlloc.max_size( );

    cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v is used up is: "
        << vsize << "." << endl;
}

```

allocator::value_type

A type that is managed by the allocator.

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.

Example

```

// allocator_value_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector <double> v;
    vector <double> ::iterator vIter, vfIter;
    vector <double> :: allocator_type vAlloc;

    int j;
    for ( j = 1 ; j <= 7 ; j++ )
    {
        v.push_back( 100.0 * j );
    }

    cout << "The original vector v is:\n ( " ;
    for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;

    vfIter = v.begin( );
    allocator<double>::value_type vecVal = 150.0;
    *vfIter = vecVal;
    cout << "The value of the element addressed by vfIter is: "
        << *vfIter << ",\n the first element in the vector." << endl;

    cout << "The modified vector v is:\n ( " ;
    for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
        cout << *vIter << " ";
    cout << ")." << endl;
}

```

```

The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element addressed by vfIter is: 150,
the first element in the vector.
The modified vector v is:
( 150 200 300 400 500 600 700 ).

```

See also

[Thread Safety in the C++ Standard Library](#)

allocator<void> Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A specialization of the template class `allocator` to type **void**, defining the types that make sense in this context.

Syntax

```
template <>
class allocator<void> {
    typedef void *pointer;
    typedef const void *const_pointer;
    typedef void value_type;
    template <class Other>
    struct rebind;
    allocator();
    allocator(const allocator<void>&);

    template <class Other>
    allocator(const allocator<Other>&);

    template <class Other>
    allocator<void>& operator=(const allocator<Other>&);
};
```

Remarks

The class explicitly specializes template class `allocator` for type **void**. Its constructors and assignment operator behave the same as for the template class, but it defines only the following types:

- `const_pointer`.
- `pointer`.
- `value_type`.
- `rebind`, a nested template class.

Requirements

Header: <memory>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

allocator_traits Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

The template class describes an object that supplements an *allocator type*. An allocator type is any type that describes an allocator object that is used for managing allocated storage. Specifically, for any allocator type `Alloc`, you can use `allocator_traits<Alloc>` to determine all the information that is needed by an allocator-enabled container. For more information, see the default [allocator Class](#).

Syntax

```
template <class Alloc>
class allocator_traits;
```

Typedefs

NAME	DESCRIPTION
<code>allocator_traits::allocator_type</code>	This type is a synonym for the template parameter <code>Alloc</code> .
<code>allocator_traits::const_pointer</code>	This type is <code>Alloc::const_pointer</code> , if that type is well-formed; otherwise, this type is <code>pointer_traits<pointer>::rebind<const value_type></code> .
<code>allocator_traits::const_void_pointer</code>	This type is <code>Alloc::const_void_pointer</code> , if that type is well-formed; otherwise, this type is <code>pointer_traits<pointer>::rebind<const void></code> .
<code>allocator_traits::difference_type</code>	This type is <code>Alloc::difference_type</code> , if that type is well-formed; otherwise, this type is <code>pointer_traits<pointer>::difference_type</code> .
<code>allocator_traits::pointer</code>	This type is <code>Alloc::pointer</code> , if that type is well-formed; otherwise, this type is <code>value_type *</code> .
<code>allocator_traits::propagate_on_container_copy_assignment</code>	This type is <code>Alloc::propagate_on_container_copy_assignment</code> , if that type is well-formed; otherwise, this type is <code>false_type</code> .
<code>allocator_traits::propagate_on_container_move_assignment</code>	This type is <code>Alloc::propagate_on_container_move_assignment</code> , if that type is well-formed; otherwise, this type is <code>false_type</code> . If the type holds true, an allocator-enabled container copies its stored allocator on a move assignment.
<code>allocator_traits::propagate_on_container_swap</code>	This type is <code>Alloc::propagate_on_container_swap</code> , if that type is well-formed; otherwise, this type is <code>false_type</code> . If the type holds true, an allocator-enabled container swaps its stored allocator on a swap.

NAME	DESCRIPTION
<code>allocator_traits::size_type</code>	This type is <code>Alloc::size_type</code> , if that type is well-formed; otherwise, this type is <code>make_unsigned<difference_type>::type</code> .
<code>allocator_traits::value_type</code>	This type is a synonym for <code>Alloc::value_type</code> .
<code>allocator_traits::void_pointer</code>	This type is <code>Alloc::void_pointer</code> , if that type is well-formed; otherwise, this type is <code>pointer_traits<pointer>::rebind<void></code> .

Static Methods

The following static methods call the corresponding method on a given allocator parameter.

NAME	DESCRIPTION
<code>allocate</code>	Static method that allocates memory by using the given allocator parameter.
<code>construct</code>	Static method that uses a specified allocator to construct an object.
<code>deallocate</code>	Static method that uses a specified allocator to deallocate a specified number of objects.
<code>destroy</code>	Static method that uses a specified allocator to call the destructor on an object without deallocating its memory.
<code>max_size</code>	Static method that uses a specified allocator to determine the maximum number of objects that can be allocated.
<code>select_on_container_copy_construction</code>	Static method that calls <code>select_on_container_copy_construction</code> on the specified allocator.

Requirements

Header: <memory>

Namespace: std

`allocator_traits::allocate`

Static method that allocates memory by using the given allocator parameter.

```
static pointer allocate(Alloc& al, size_type count);

static pointer allocate(Alloc& al, size_type count,
    typename allocator_traits<void>::const_pointer* hint);
```

Parameters

al

An allocator object.

count

The number of elements to allocate.

hint

A `const_pointer` that might assist the allocator object in satisfying the request for storage by locating the address of an allocated object prior to the request. A null pointer is treated as no hint.

Return Value

Each method returns a pointer to the allocated object.

The first static method returns `a1.allocate(count)`.

The second method returns `a1.allocate(count, hint)`, if that expression is well formed; otherwise it returns `a1.allocate(count)`.

allocator_traits::construct

Static method that uses a specified allocator to construct an object.

```
template <class Uty, class Types>
static void construct(Alloc& al, Uty* ptr, Types&&... args);
```

Parameters

al

An allocator object.

ptr

A pointer to the location where the object is to be constructed.

args

A list of arguments that is passed to the object constructor.

Remarks

The static member function calls `a1.construct(ptr, args...)`, if that expression is well formed; otherwise it evaluates `::new (static_cast<void *>(ptr)) Uty(std::forward<Types>(args)...) .`

allocator_traits::deallocate

Static method that uses a specified allocator to deallocate a specified number of objects.

```
static void deallocate(Alloc al,
    pointer ptr,
    size_type count);
```

Parameters

al

An allocator object.

ptr

A pointer to the starting location of the objects to be deallocated.

count

The number of objects to deallocate.

Remarks

This method calls `al.deallocate(ptr, count)`.

This method throws nothing.

allocator_traits::destroy

Static method that uses a specified allocator to call the destructor on an object without deallocating its memory.

```
template <class Uty>
static void destroy(Alloc& al, Uty* ptr);
```

Parameters

al

An allocator object.

ptr

A pointer to the location of the object.

Remarks

This method calls `al.destroy(ptr)`, if that expression is well formed; otherwise it evaluates `ptr->~Uty()`.

allocator_traits::max_size

Static method that uses a specified allocator to determine the maximum number of objects that can be allocated.

```
static size_type max_size(const Alloc& al);
```

Parameters

al

An allocator object.

Remarks

This method returns `al.max_size()`, if that expression is well formed; otherwise it returns `numeric_limits<size_type>::max()`.

allocator_traits::select_on_container_copy_construction

Static method that calls `select_on_container_copy_construction` on the specified allocator.

```
static Alloc select_on_container_copy_construction(const Alloc& al);
```

Parameters

al

An allocator object.

Return Value

This method returns `al.select_on_container_copy_construction()`, if that type is well formed; otherwise it returns *al*.

Remarks

This method is used to specify an allocator when the associated container is copy-constructed.

See also

[<memory>](#)

[pointer_traits](#) Struct

[scoped_allocator_adaptor](#) Class

auto_ptr Class

11/8/2018 • 7 minutes to read • [Edit Online](#)

Wraps a smart pointer around a resource that ensures the resource is destroyed automatically when control leaves a block.

The more capable `unique_ptr` class supersedes `auto_ptr`. For more information, see [unique_ptr Class](#).

For more information about `throw()` and exception handling, see [Exception Specifications \(throw\)](#).

Syntax

```
class auto_ptr {
public:
    typedef Type element_type;
    explicit auto_ptr(Type* ptr = 0) throw();
    auto_ptr(auto_ptr<Type>& right) throw()
        ;
    template <class Other>
    operator auto_ptr<Other>() throw();
    template <class Other>
    auto_ptr<Type>& operator=(auto_ptr<Other>& right) throw();
    template <class Other>
    auto_ptr(auto_ptr<Other>& right);
    auto_ptr<Type>& operator=(auto_ptr<Type>& right);
    ~auto_ptr();
    Type& operator*() const throw();
    Type * operator->()const throw();
    Type *get() const throw();
    Type *release()throw();
    void reset(Type* ptr = 0);
};
```

Parameters

right

The `auto_ptr` from which to get an existing resource.

ptr

The pointer specified to replace the stored pointer.

Remarks

The template class describes a smart pointer, called an `auto_ptr`, to an allocated object. The pointer must be either null or designate an object allocated by **new**. The `auto_ptr` transfers ownership if its stored value is assigned to another object. (It replaces the stored value after a transfer with a null pointer.) The destructor for `auto_ptr<Type>` deletes the allocated object. The `auto_ptr<Type>` ensures that an allocated object is automatically deleted when control leaves a block, even through a thrown exception. You should not construct two `auto_ptr<Type>` objects that own the same object.

You can pass an `auto_ptr<Type>` object by value as an argument to a function call. An `auto_ptr` cannot be an element of any Standard Library container. You cannot reliably manage a sequence of `auto_ptr<Type>` objects with a C++ Standard Library container.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
<code>auto_ptr</code>	The constructor for objects of type <code>auto_ptr</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>element_type</code>	The type is a synonym for the template parameter <code>Type</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>get</code>	The member function returns the stored pointer <code>myptr</code> .
<code>release</code>	The member replaces the stored pointer <code>myptr</code> with a null pointer and returns the previously stored pointer.
<code>reset</code>	The member function evaluates the expression <code>delete myptr</code> , but only if the stored pointer value <code>myptr</code> changes as a result of function call. It then replaces the stored pointer with <i>ptr</i> .

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	An assignment operator that transfers ownership from one <code>auto_ptr</code> object to another.
<code>operator*</code>	The dereferencing operator for objects of type <code>auto_ptr</code> .
<code>operator-></code>	The operator for allowing member access.
<code>operator auto_ptr<Other></code>	Casts from one kind of <code>auto_ptr</code> to another kind of <code>auto_ptr</code> .
<code>operator auto_ptr_ref<Other></code>	Casts from an <code>auto_ptr</code> to an <code>auto_ptr_ref</code> .

Requirements

Header: `<memory>`

Namespace: `std`

`auto_ptr::auto_ptr`

The constructor for objects of type `auto_ptr`.

```
explicit auto_ptr(Type* ptr = 0) throw();

auto_ptr(auto_ptr<Type>& right) throw();

auto_ptr(auto_ptr_ref<Type> right) throw();

template <class Other>
auto_ptr(auto_ptr<Other>& right) throw();
```

Parameters

ptr

The pointer to the object that `auto_ptr` encapsulates.

right

The `auto_ptr` object to be copied by the constructor.

Remarks

The first constructor stores *ptr* in `myptr`, the stored pointer to the allocated object. The second constructor transfers ownership of the pointer stored in *right*, by storing *right.release* in `myptr`.

The third constructor behaves the same as the second, except that it stores `right.ref.release` in `myptr`, where `ref` is the reference stored in `right`.

The template constructor behaves the same as the second constructor, provided that a pointer to `Other` can be implicitly converted to a pointer to `Type`.

Example

```

// auto_ptr_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
    Int(int i)
    {
        cout << "Constructing " << ( void* )this << endl;
        x = i;
        bIsConstructed = true;
    };
    ~Int( )
    {
        cout << "Destructing " << ( void* )this << endl;
        bIsConstructed = false;
    };
    Int &operator++( )
    {
        x++;
        return *this;
    };
    int x;
private:
    bool bIsConstructed;
};

void function ( auto_ptr<Int> &pi )
{
    ++( *pi );
    auto_ptr<Int> pi2( pi );
    ++( *pi2 );
    pi = pi2;
}

int main( )
{
    auto_ptr<Int> pi ( new Int( 5 ) );
    cout << pi->x << endl;
    function( pi );
    cout << pi->x << endl;
}

```

```

Constructing 00311AF8
5
7
Destructing 00311AF8

```

auto_ptr::element_type

The type is a synonym for the template parameter `Type`.

```
typedef Type element _type;
```


auto_ptr::get

The member function returns the stored pointer `myptr`.

```
Type *get() const throw();
```

Return Value

The stored pointer `myptr`.

Example

```
// auto_ptr_get.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

class Int
{
public:
    Int(int i)
    {
        x = i;
        cout << "Constructing " << ( void* )this << " Value: " << x << endl;
    };
    ~Int( )
    {
        cout << "Destructing " << ( void* )this << " Value: " << x << endl;
    };

    int x;
};

int main( )
{
    auto_ptr<Int> pi ( new Int( 5 ) );
    pi.reset( new Int( 6 ) );
    Int* pi2 = pi.get ( );
    Int* pi3 = pi.release ( );
    if (pi2 == pi3)
        cout << "pi2 == pi3" << endl;
    delete pi3;
}
```

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

auto_ptr::operator=

An assignment operator that transfers ownership from one `auto_ptr` object to another.

```
template <class Other>
auto_ptr<Type>& operator=(auto_ptr<Other>& right) throw();
auto_ptr<Type>& operator=(auto_ptr<Type>& right) throw();
auto_ptr<Type>& operator=(auto_ptr_ref<Type> right) throw();
```

Parameters

right

An object of type `auto_ptr`.

Return Value

A reference to an object of type `auto_ptr<Type>`.

Remarks

The assignment evaluates the expression `delete myptr`, but only if the stored pointer `myptr` changes as a result of the assignment. It then transfers ownership of the pointer stored in *right*, by storing *right.release* in `myptr`. The function returns ***this**.

Example

For an example of the use of the member operator, see [auto_ptr::auto_ptr](#).

auto_ptr::operator*

The dereferencing operator for objects of type `auto_ptr`.

```
Type& operator*() const throw();
```

Return Value

A reference to an object of type `Type` that the pointer owns.

Remarks

The indirection operator returns `*get`. Hence, the stored pointer must not be null.

Example

For an example of how to use the member function, see [auto_ptr::auto_ptr](#).

auto_ptr::operator->

The operator for allowing member access.

```
Type * operator->() const throw();
```

Return Value

A member of the object that `auto_ptr` owns.

Remarks

The selection operator returns `get ()`, so that the expression *ap*-> **member** behaves the same as (*ap*. **get**())-> **member**, where *ap* is an object of class `auto_ptr < Type>`. Hence, the stored pointer must not be null, and `Type` must be a class, struct, or union type with a `member` member.

Example

For an example of how to use the member function, see [auto_ptr::auto_ptr](#).

auto_ptr::operator auto_ptr<Other>

Casts from one kind of `auto_ptr` to another kind of `auto_ptr`.

```
template <class Other>
operator auto_ptr<Other>() throw();
```

Return Value

The type cast operator returns `auto_ptr < Other> (*this)`.

Example

```
// auto_ptr_op_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;
int main()
{
    auto_ptr<int> pi ( new int( 5 ) );
    auto_ptr<const int> pc = ( auto_ptr<const int> )pi;
}
```

auto_ptr::operator auto_ptr_ref<Other>

Casts from an `auto_ptr` to an `auto_ptr_ref`.

```
template <class Other>
operator auto_ptr_ref<Other>() throw();
```

Return Value

The type cast operator returns `auto_ptr_ref< Other> (*this)`.

Example

```

// auto_ptr_op_auto_ptr_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class C {
public:
    C(int _i) : m_i(_i) {
    }
    ~C() {
        cout << "~C:  " << m_i << "\n";
    }
    C &operator =(const int &x) {
        m_i = x;
        return *this;
    }
    int m_i;
};

void f(auto_ptr<C> arg) {
};

int main()
{
    const auto_ptr<C> ciap(new C(1));
    auto_ptr<C> iap(new C(2));

    // Error: this implies transfer of ownership of iap's pointer
    // f(ciap);
    f(iap); // compiles, but gives up ownership of pointer

            // here, iap owns a destroyed pointer so the following is bad:
            // *iap = 5; // BOOM

    cout << "main exiting\n";
}

```

```

~C:  2
main exiting
~C:  1

```

auto_ptr::release

The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.

```
Type *release() throw();
```

Return Value

The previously stored pointer.

Remarks

The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.

Example

```

// auto_ptr_release.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

class Int
{
public:
    Int(int i)
    {
        x = i;
        cout << "Constructing " << (void*)this << " Value: " << x << endl;
    };
    ~Int() {
        cout << "Destructing " << (void*)this << " Value: " << x << endl;
    };

    int x;
};

int main()
{
    auto_ptr<Int> pi(new Int(5));
    pi.reset(new Int(6));
    Int* pi2 = pi.get();
    Int* pi3 = pi.release();
    if (pi2 == pi3)
        cout << "pi2 == pi3" << endl;
    delete pi3;
}

```

```

Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6

```

auto_ptr::reset

The member function evaluates the expression `delete myptr`, but only if the stored pointer value `myptr` changes as a result of a function call. It then replaces the stored pointer with `ptr`.

```
void reset(Type* ptr = 0);
```

Parameters

ptr

The pointer specified to replace the stored pointer `myptr`.

Example

```

// auto_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
    Int(int i)
    {
        x = i;
        cout << "Constructing " << (void*)this << " Value: " << x << endl;
    };
    ~Int()
    {
        cout << "Destructing " << (void*)this << " Value: " << x << endl;
    };

    int x;
};

int main()
{
    auto_ptr<Int> pi(new Int(5));
    pi.reset(new Int(6));
    Int* pi2 = pi.get();
    Int* pi3 = pi.release();
    if (pi2 == pi3)
        cout << "pi2 == pi3" << endl;
    delete pi3;
}

```

```

Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6

```

See also

[Thread Safety in the C++ Standard Library](#)

[unique_ptr Class](#)

bad_weak_ptr Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Reports bad weak_ptr exception.

Syntax

```
class bad_weak_ptr : public std::exception
{
public:
    bad_weak_ptr();
    const char *what() throw();
};
```

Remarks

The class describes an exception that can be thrown from the [shared_ptr Class](#) constructor that takes an argument of type [weak_ptr Class](#). The member function `what` returns `"bad_weak_ptr"`.

Example

```
// std__memory__bad_weak_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int);
        wp = sp;
    }

    try
    {
        std::shared_ptr<int> sp1(wp); // weak_ptr has expired
    }
    catch (const std::bad_weak_ptr&)
    {
        std::cout << "bad weak pointer" << std::endl;
    }
    catch (...)
    {
        std::cout << "unknown exception" << std::endl;
    }

    return (0);
}
```

bad weak pointer

Requirements

Header: <memory>

Namespace: std

See also

[weak_ptr](#) Class

enable_shared_from_this Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Helps generate a `shared_ptr`.

Syntax

```
class enable_shared_from_this {
public:
    shared_ptr<Ty>
        shared_from_this();
    shared_ptr<const Ty> shared_from_this() const;
protected:
    enable_shared_from_this();
    enable_shared_from_this(const enable_shared_from_this&);
    enable_shared_from_this& operator=(const enable_shared_from_this&);
    ~enable_shared_from_this();
};
```

Parameters

Ty

The type controlled by the shared pointer.

Remarks

Objects derived from `enable_shared_from_this` can use the `shared_from_this` methods in member functions to create `shared_ptr` owners of the instance that share ownership with existing `shared_ptr` owners. Otherwise, if you create a new `shared_ptr` by using **this**, it is distinct from existing `shared_ptr` owners, which can lead to invalid references or cause the object to be deleted more than once.

The constructors, destructor, and assignment operator are protected to help prevent accidental misuse. The template argument type *Ty* must be the type of the derived class.

For an example of usage, see [enable_shared_from_this::shared_from_this](#).

Requirements

Header: <memory>

Namespace: std

enable_shared_from_this::shared_from_this

Generates a `shared_ptr` that shares ownership of the instance with existing `shared_ptr` owners.

```
shared_ptr<T> shared_from_this();
shared_ptr<const T> shared_from_this() const;
```

Remarks

When you derive objects from the `enable_shared_from_this` base class, the `shared_from_this` template member functions return a `shared_ptr Class` object that shares ownership of this instance with existing `shared_ptr` owners.

Otherwise, if you create a new `shared_ptr` from **this**, it is distinct from existing `shared_ptr` owners, which can lead to invalid references or cause the object to be deleted more than once. The behavior is undefined if you call `shared_from_this` on an instance that is not already owned by a `shared_ptr` object.

Example

```
// std_memory_shared_from_this.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

struct base : public std::enable_shared_from_this<base>
{
    int val;
    shared_ptr<base> share_more()
    {
        return shared_from_this();
    }
};

int main()
{
    auto sp1 = make_shared<base>();
    auto sp2 = sp1->share_more();

    sp1->val = 3;
    cout << "sp2->val == " << sp2->val << endl;
    return 0;
}
```

```
sp2->val == 3
```

See also

[enable_shared_from_this::shared_from_this](#)

[shared_ptr Class](#)

pointer_traits Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

Supplies information that is needed by an object of template class `allocator_traits` to describe an allocator with pointer type `Ptr`.

Syntax

```
template <class Ptr>
struct pointer_traits;
```

Remarks

`Ptr` can be a raw pointer of type `Ty *` or a class with the following properties.

```
struct Ptr
{ // describes a pointer type usable by allocators
    typedef Ptr pointer;
    typedef T1 element_type; // optional
    typedef T2 difference_type; // optional
    template <class Other>
    using rebind = typename Ptr<Other, Rest...>; // optional
    static pointer pointer_to(element_type& obj);
    // optional
};
```

Typedefs

NAME	DESCRIPTION
<code>typedef T2 difference_type</code>	The type <code>T2</code> is <code>Ptr::difference_type</code> if that type exists, otherwise <code>ptrdiff_t</code> . If <code>Ptr</code> is a raw pointer, the type is <code>ptrdiff_t</code> .
<code>typedef T1 element_type</code>	The type <code>T1</code> is <code>Ptr::element_type</code> if that type exists, otherwise <code>Ty</code> . If <code>Ptr</code> is a raw pointer, the type is <code>Ty</code> .
<code>typedef Ptr pointer</code>	The type is <code>Ptr</code> .

Structs

NAME	DESCRIPTION
<code>pointer_traits::rebind</code>	Attempts to convert the underlying pointer type to a specified type.

Methods

NAME	DESCRIPTION
pointer_to	Converts an arbitrary reference to an object of class <code>Ptr</code> .

Requirements

Header: `<memory>`

Namespace: `std`

pointer_to

Static method that returns `Ptr::pointer_to(obj)`, if that function exists. Otherwise, it is not possible to convert an arbitrary reference to an object of class `Ptr`. If `Ptr` is a raw pointer, this method returns `addressof(obj)`.

```
static pointer pointer_to(element_type& obj);
```

See also

[<memory>](#)

[allocator_traits](#) Class

raw_storage_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.

Syntax

```
template <class OutputIterator, class Type>
class raw_storage_iterator
```

Parameters

OutputIterator

Specifies the output iterator for the object being stored.

Type

The type of object for which storage is being allocated.

Remarks

The class describes an output iterator that constructs objects of type `Type` in the sequence it generates. An object of class `raw_storage_iterator` < **ForwardIterator**, **Type**> accesses storage through a forward iterator object, of class `ForwardIterator`, that you specify when you construct the object. For an object first of class `ForwardIterator`, the expression **&*first** must designate unconstructed storage for the next object (of type `Type`) in the generated sequence.

This adaptor class is used when it is necessary to separate memory allocation and object construction. The `raw_storage_iterator` can be used to copy objects into uninitialized storage, such as memory allocated using the `malloc` function.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
<code>raw_storage_iterator</code>	Constructs a raw storage iterator with a specified underlying output iterator.

Typedefs

TYPE NAME	DESCRIPTION
<code>element_type</code>	Provides a type that describes an element to be stored a raw storage iterator.
<code>iter_type</code>	Provides a type that describes an iterator that underlies a raw storage iterator.

Operators

OPERATOR	DESCRIPTION
<code>operator*</code>	A dereferencing operator used to implement the output iterator expression <code>* ii = x</code> .
<code>operator=</code>	An assignment operator used to implement the raw storage iterator expression <code>* i = x</code> for storing in memory.
<code>operator++</code>	Preincrement and postincrement operators for raw storage iterators.

Requirements

Header: <memory>

Namespace: std

raw_storage_iterator::element_type

Provides a type that describes an element to be stored a raw storage iterator.

```
typedef Type element_type;
```

Remarks

The type is a synonym for the raw_storage_iterator class template parameter `Type`.

raw_storage_iterator::iter_type

Provides a type that describes an iterator that underlies a raw storage iterator.

```
typedef ForwardIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `ForwardIterator`.

raw_storage_iterator::operator*

A dereferencing operator used to implement the raw storage iterator expression `* ii = x`.

```
raw_storage_iterator<ForwardIterator, Type>& operator*();
```

Return Value

A reference to the raw storage iterator

Remarks

The requirements for a `ForwardIterator` are that the raw storage iterator must satisfy require only the expression `* ii = t` be valid and that it says nothing about the **operator** or the `operator=` on their own. The member operators in this implementation returns ***this**, so that `operator=(constType&)` can perform the actual store in an expression, such as `* ptr = val`.

Example

```

// raw_storage_iterator_op_deref.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
    Int(int i)
    {
        cout << "Constructing " << i << endl;
        x = i;
        bIsConstructed = true;
    };

    Int &operator=(int i)
    {
        if (!bIsConstructed)
            cout << "Not constructed.\n";
        cout << "Copying " << i << endl;
        x = i;
        return *this;
    };

    int x;

private:
    bool bIsConstructed;
};

int main( void)
{
    Int *pInt = ( Int* ) malloc( sizeof( Int ) );
    memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;
    *pInt = 5;
    raw_storage_iterator< Int*, Int > it( pInt );
    *it = 5;
}
/* Output:
Not constructed.
Copying 5
Constructing 5
*/

```

raw_storage_iterator::operator=

Assignment operator used to implement the raw storage iterator expression $*i = x$ for storing in memory.

```

raw_storage_iterator<ForwardIterator, Type>& operator=(
    const Type& val);

```

Parameters

val

The value of the object of type `Type` to be inserted into memory.

Return Value

The operator inserts `val` into memory, and then returns a reference to the raw storage iterator.

Remarks

The requirements for a `ForwardIterator` state that the raw storage iterator must satisfy require only the expression `*i = t` be valid, and that it says nothing about the **operator** or the `operator=` on their own. These member operators return ***this**.

The assignment operator constructs the next object in the output sequence using the stored iterator value first, by evaluating the placement new expression `new ((void *)&* first) Type(val)`.

Example

```
// raw_storage_iterator_op_assign.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
    Int( int i )
    {
        cout << "Constructing " << i << endl;
        x = i;
        bIsConstructed = true;
    };
    Int &operator=( int i )
    {
        if ( !bIsConstructed )
            cout << "Not constructed.\n";
        cout << "Copying " << i << endl; x = i;
        return *this;
    };
    int x;
private:
    bool bIsConstructed;
};

int main( void )
{
    Int *pInt = ( Int* )malloc( sizeof( Int ) );
    memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;

    *pInt = 5;

    raw_storage_iterator<Int*, Int> it( pInt );
    *it = 5;
}
/* Output:
Not constructed.
Copying 5
Constructing 5
*/
```

raw_storage_iterator::operator++

Preincrement and postincrement operators for raw storage iterators.

```
raw_storage_iterator<ForwardIterator, Type>& operator++();

raw_storage_iterator<ForwardIterator, Type> operator++(int);
```


Return Value

An raw storage iterator or a reference to an raw storage iterator.

Remarks

The first operator eventually attempts to extract and store an object of type `CharType` from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

The first preincrement operator increments the stored output iterator object, and then returns ***this**.

The second postincrement operator makes a copy of ***this**, increments the stored output iterator object, and then returns the copy.

The constructor stores `first` as the output iterator object.

Example

```
// raw_storage_iterator_op_incr.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

int main( void )
{
    int *pInt = new int[5];
    std::raw_storage_iterator<int*,int> it( pInt );
    for ( int i = 0; i < 5; i++, it++ ) {
        *it = 2 * i;
    };

    for ( int i = 0; i < 5; i++ ) cout << "array " << i << " = " << pInt[i] << endl;;

    delete[] pInt;
}
/* Output:
array 0 = 0
array 1 = 2
array 2 = 4
array 3 = 6
array 4 = 8
*/
```

raw_storage_iterator::raw_storage_iterator

Constructs a raw storage iterator with a specified underlying output iterator.

```
explicit raw_storage_iterator(ForwardIterator first);
```

Parameters

first

The forward iterator that is to underlie the `raw_storage_iterator` object being constructed.

Example

```
// raw_storage_iterator_ctor.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <iterator>
#include <memory>
```

```

#include <memory>
#include <list>
using namespace std;

class Int
{
public:
    Int(int i)
    {
        cout << "Constructing " << i << endl;
        x = i;
        bIsConstructed = true;
    };
    Int &operator=( int i )
    {
        if (!bIsConstructed)
            cout << "Error! I'm not constructed!\n";
        cout << "Copying " << i << endl; x = i; return *this;
    };
    int x;
    bool bIsConstructed;
};

int main( void )
{
    std::list<int> l;
    l.push_back( 1 );
    l.push_back( 2 );
    l.push_back( 3 );
    l.push_back( 4 );

    Int *pInt = (Int*)malloc(sizeof(Int)*l.size( ));
    memset (pInt, 0, sizeof(Int)*l.size( ));
    // Hack: make sure bIsConstructed is false

    std::copy( l.begin( ), l.end( ), pInt ); // C4996
    for (unsigned int i = 0; i < l.size( ); i++)
        cout << "array " << i << " = " << pInt[i].x << endl;;

    memset (pInt, 0, sizeof(Int)*l.size( ));
    // hack: make sure bIsConstructed is false

    std::copy( l.begin( ), l.end( ),
        std::raw_storage_iterator<Int*,Int>(pInt)); // C4996
    for (unsigned int i = 0; i < l.size( ); i++ )
        cout << "array " << i << " = " << pInt[i].x << endl;

    free(pInt);
}
/* Output:
Error! I'm not constructed!
Copying 1
Error! I'm not constructed!
Copying 2
Error! I'm not constructed!
Copying 3
Error! I'm not constructed!
Copying 4
array 0 = 1
array 1 = 2
array 2 = 3
array 3 = 4
Constructing 1
Constructing 2
Constructing 3
Constructing 4
array 0 = 1
array 1 = 2
array 2 = 3

```

```
array 3 = 4  
*/
```

See also

[Thread Safety in the C++ Standard Library](#)

shared_ptr Class

11/9/2018 • 12 minutes to read • [Edit Online](#)

Wraps a reference-counted smart pointer around a dynamically allocated object.

Syntax

```
template <class T>
class shared_ptr;
```

Remarks

The `shared_ptr` class describes an object that uses reference counting to manage resources. A `shared_ptr` object effectively holds a pointer to the resource that it owns or holds a null pointer. A resource can be owned by more than one `shared_ptr` object; when the last `shared_ptr` object that owns a particular resource is destroyed, the resource is freed.

A `shared_ptr` stops owning a resource when it is reassigned or reset.

The template argument `T` might be an incomplete type except as noted for certain member functions.

When a `shared_ptr<T>` object is constructed from a resource pointer of type `G*` or from a `shared_ptr<G>`, the pointer type `G*` must be convertible to `T*`. If it is not, the code will not compile. For example:

```
#include <memory>
using namespace std;

class F {};
class G : public F {};

shared_ptr<G> sp0(new G); // okay, template parameter G and argument G*
shared_ptr<G> sp1(sp0);  // okay, template parameter G and argument shared_ptr<G>
shared_ptr<F> sp2(new G); // okay, G* convertible to F*
shared_ptr<F> sp3(sp0);  // okay, template parameter F and argument shared_ptr<G>
shared_ptr<F> sp4(sp2);  // okay, template parameter F and argument shared_ptr<F>
shared_ptr<int> sp5(new G); // error, G* not convertible to int*
shared_ptr<int> sp6(sp2);  // error, template parameter int and argument shared_ptr<F>
```

A `shared_ptr` object owns a resource:

- if it was constructed with a pointer to that resource,
- if it was constructed from a `shared_ptr` object that owns that resource,
- if it was constructed from a [weak_ptr Class](#) object that points to that resource, or
- if ownership of that resource was assigned to it, either with `shared_ptr::operator=` or by calling the member function `shared_ptr::reset`.

The `shared_ptr` objects that own a resource share a control block. The control block holds:

- the number of `shared_ptr` objects that own the resource,
- the number of `weak_ptr` objects that point to the resource,

- the deleter for that resource if it has one,
- the custom allocator for the control block if it has one.

A `shared_ptr` object that is initialized by using a null pointer has a control block and is not empty. After a `shared_ptr` object releases a resource, it no longer owns that resource. After a `weak_ptr` object releases a resource, it no longer points to that resource.

When the number of `shared_ptr` objects that own a resource becomes zero, the resource is freed, either by deleting it or by passing its address to a deleter, depending on how ownership of the resource was originally created. When the number of `shared_ptr` objects that own a resource is zero, and the number of `weak_ptr` objects that point to that resource is zero, the control block is freed, using the custom allocator for the control block if it has one.

An empty `shared_ptr` object does not own any resources and has no control block.

A deleter is a function object that has a member function `operator()`. Its type must be copy constructible, and its copy constructor and destructor must not throw exceptions. It accepts one parameter, the object to be deleted.

Some functions take an argument list that defines properties of the resulting `shared_ptr<T>` or `weak_ptr<T>` object. You can specify such an argument list in several ways:

no arguments -- the resulting object is an empty `shared_ptr` object or an empty `weak_ptr` object.

`ptr` -- a pointer of type `other*` to the resource to be managed. `T` must be a complete type. If the function fails (because the control block cannot be allocated) it evaluates the expression `delete ptr`.

`ptr, dtor` -- a pointer of type `other*` to the resource to be managed and a deleter for that resource. If the function fails (because the control block cannot be allocated), it calls `dtor(ptr)`, which must be well defined.

`ptr, dtor, alloc` -- a pointer of type `other*` to the resource to be managed, a deleter for that resource, and an allocator to manage any storage that must be allocated and freed. If the function fails (because the control block can't be allocated) it calls `dtor(ptr)`, which must be well defined.

`sp` -- a `shared_ptr<Other>` object that owns the resource to be managed.

`wp` -- a `weak_ptr<Other>` object that points to the resource to be managed.

`ap` -- an `auto_ptr<Other>` object that holds a pointer to the resource to be managed. If the function succeeds it calls `ap.release()`; otherwise it leaves `ap` unchanged.

In all cases, the pointer type `other*` must be convertible to `T*`.

Thread Safety

Multiple threads can read and write different `shared_ptr` objects at the same time, even when the objects are copies that share ownership.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
<code>shared_ptr</code>	Constructs a <code>shared_ptr</code> .
<code>shared_ptr::~~shared_ptr</code>	Destroys a <code>shared_ptr</code> .

Types

TYPE NAME	DESCRIPTION
<code>element_type</code>	The type of an element.

Functions

FUNCTION	DESCRIPTION
<code>get</code>	Gets address of owned resource.
<code>owner_before</code>	Returns true if this <code>shared_ptr</code> is ordered before (or less than) the provided pointer.
<code>reset</code>	Replace owned resource.
<code>swap</code>	Swaps two <code>shared_ptr</code> objects.
<code>unique</code>	Tests if owned resource is unique.
<code>use_count</code>	Counts numbers of resource owners.

Operators

OPERATOR	DESCRIPTION
<code>shared_ptr::operator bool</code>	Tests if an owned resource exists.
<code>shared_ptr::operator*</code>	Gets the designated value.
<code>shared_ptr::operator=</code>	Replaces the owned resource.
<code>shared_ptr::operator-></code>	Gets a pointer to the designated value.

Requirements

Header: <memory>

Namespace: std

`shared_ptr::element_type`

The type of an element.

```
typedef T element_type;
```

Remarks

The type is a synonym for the template parameter `T`.

Example

```
// std__memory__shared_ptr_element_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::shared_ptr<int>::element_type val = *sp0;

    std::cout << "*sp0 == " << val << std::endl;

    return (0);
}
```

```
*sp0 == 5
```

shared_ptr::get

Gets address of owned resource.

```
T *get() const;
```

Remarks

The member function returns the address of the owned resource. If the object does not own a resource it returns 0.

Example

```
// std__memory__shared_ptr_get.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));

    std::cout << "sp0.get() == 0 == " << std::boolalpha
              << (sp0.get() == 0) << std::endl;
    std::cout << "*sp1.get() == " << *sp1.get() << std::endl;

    return (0);
}
```

```
sp0.get() == 0 == true
*sp1.get() == 5
```

shared_ptr::operator bool

Tests if an owned resource exists.

```
explicit operator bool() const noexcept;
```

Remarks

The operator returns a value of **true** when `get() != nullptr`, otherwise **false**.

Example

```
// std__memory__shared_ptr_operator_bool.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));

    std::cout << "(bool)sp0 == " << std::boolalpha
        << (bool)sp0 << std::endl;
    std::cout << "(bool)sp1 == " << std::boolalpha
        << (bool)sp1 << std::endl;

    return (0);
}
```

```
(bool)sp0 == false
(bool)sp1 == true
```

shared_ptr::operator*

Gets the designated value.

```
T& operator*() const;
```

Remarks

The indirection operator returns `*get()`. Hence, the stored pointer must not be null.

Example

```
// std__memory__shared_ptr_operator_st.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));

    std::cout << "*sp0 == " << *sp0 << std::endl;

    return (0);
}
```

```
*sp0 == 5
```

shared_ptr::operator=

Replaces the owned resource.


```

shared_ptr& operator=(const shared_ptr& sp);

template <class Other>
shared_ptr& operator=(const shared_ptr<Other>& sp);

template <class Other>
shared_ptr& operator=(auto_ptr<Other>& ap);

template <class Other>
shared_ptr& operator=(auto_ptr<Other>& ap);

template <class Other>
shared_ptr& operator=(auto_ptr<Other>&& ap);

template <class Other, class Deletor>
shared_ptr& operator=(unique_ptr<Other, Deletor>&& ap);

```

Parameters

sp

The shared pointer to copy.

ap

The auto pointer to copy.

Remarks

The operators all decrement the reference count for the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If the reference count falls to zero, the resource is released. If an operator fails it leaves `*this` unchanged.

Example

```

// std__memory__shared_ptr_operator_as.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));
    std::auto_ptr<int> ap(new int(10));

    sp0 = sp1;
    std::cout << "sp0 == " << *sp0 << std::endl;

    sp0 = ap;
    std::cout << "sp0 == " << *sp0 << std::endl;

    return (0);
}

```

```

*sp0 == 5
*sp0 == 10

```

shared_ptr::operator->

Gets a pointer to the designated value.

```
T * operator->() const;
```

Remarks

The selection operator returns `get()`, so that the expression `sp->member` behaves the same as `(sp.get())->member` where `sp` is an object of class `shared_ptr<T>`. Hence, the stored pointer must not be null, and `T` must be a class, structure, or union type with a member `member`.

Example

```
// std__memory__shared_ptr_operator_ar.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

typedef std::pair<int, int> Mypair;
int main()
{
    std::shared_ptr<Mypair> sp0(new Mypair(1, 2));

    std::cout << "sp0->first == " << sp0->first << std::endl;
    std::cout << "sp0->second == " << sp0->second << std::endl;

    return (0);
}
```

```
sp0->first == 1
sp0->second == 2
```

shared_ptr::owner_before

Returns true if this `shared_ptr` is ordered before (or less than) the provided pointer.

```
template <class Other>
bool owner_before(const shared_ptr<Other>& ptr);

template <class Other>
bool owner_before(const weak_ptr<Other>& ptr);
```

Parameters

ptr

An `lvalue` reference to either a `shared_ptr` or a `weak_ptr`.

Remarks

The template member function returns true if `*this` is ordered before `ptr`.

shared_ptr::reset

Replace owned resource.

```
void reset();

template <class Other>
void reset(Other *ptr);

template <class Other, class D>
void reset(Other *ptr, D dtor);

template <class Other, class D, class A>
void reset(Other *ptr, D dtor, A alloc);
```

Parameters

Other

The type controlled by the argument pointer.

D

The type of the deleter.

ptr

The pointer to copy.

dtor

The deleter to copy.

A

The type of the allocator.

alloc

The allocator to copy.

Remarks

The operators all decrement the reference count for the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If the reference count falls to zero, the resource is released. If an operator fails it leaves `*this` unchanged.

Example

```

// std__memory__shared_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp(new int(5));

    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;

    sp.reset();
    std::cout << "(bool)sp == " << std::boolalpha
        << (bool)sp << std::endl;

    sp.reset(new int(10));
    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;

    sp.reset(new int(15), deleter());
    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;

    return (0);
}

```

```

*sp == 5
(bool)sp == false
*sp == 10
*sp == 15

```

shared_ptr::shared_ptr

Constructs a `shared_ptr` .

```

shared_ptr();

shared_ptr(nullptr_t);

shared_ptr(const shared_ptr& sp);

shared_ptr(shared_ptr&& sp);

template <class Other>
explicit shared_ptr(Other* ptr);

template <class Other, class D>
shared_ptr(Other* ptr, D dtor);

template <class D>
shared_ptr(nullptr_t ptr, D dtor);

template <class Other, class D, class A>
shared_ptr(Other* ptr, D dtor, A alloc);

template <class D, class A>
shared_ptr(nullptr_t ptr, D dtor, A alloc);

template <class Other>
shared_ptr(const shared_ptr<Other>& sp);

template <class Other>
shared_ptr(const weak_ptr<Other>& wp);

template <class >
shared_ptr(std::auto_ptr<Other>& ap);

template <class >
shared_ptr(std::auto_ptr<Other>&& ap);

template <class Other, class D>
shared_ptr(unique_ptr<Other, D>&& up);

template <class Other>
shared_ptr(const shared_ptr<Other>& sp, T* ptr);

template <class Other, class D>
shared_ptr(const unique_ptr<Other, D>& up) = delete;

```

Parameters

Other

The type controlled by the argument pointer.

ptr

The pointer to copy.

D

The type of the deleter.

A

The type of the allocator.

dtor

The deleter.

ator

The allocator.

sp

The smart pointer to copy.

wp

The weak pointer.

ap

The auto pointer to copy.

Remarks

The constructors each construct an object that owns the resource named by the operand sequence. The constructor `shared_ptr(const weak_ptr<Other>& wp)` throws an exception object of type [bad_weak_ptr Class](#) if `wp.expired()` .

Example

```
// std__memory__shared_ptr_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp0;
    std::cout << "(bool)sp0 == " << std::boolalpha
        << (bool)sp0 << std::endl;

    std::shared_ptr<int> sp1(new int(5));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    std::shared_ptr<int> sp2(new int(10), deleter());
    std::cout << "*sp2 == " << *sp2 << std::endl;

    std::shared_ptr<int> sp3(sp2);
    std::cout << "*sp3 == " << *sp3 << std::endl;

    std::weak_ptr<int> wp(sp3);
    std::shared_ptr<int> sp4(wp);
    std::cout << "*sp4 == " << *sp4 << std::endl;

    std::auto_ptr<int> ap(new int(15));
    std::shared_ptr<int> sp5(ap);
    std::cout << "*sp5 == " << *sp5 << std::endl;

    return (0);
}
```

```
(bool)sp0 == false
*sp1 == 5
*sp2 == 10
*sp3 == 10
*sp4 == 10
*sp5 == 15
```

shared_ptr::~~shared_ptr

Destroys a `shared_ptr`.

```
~shared_ptr();
```

Remarks

The destructor decrements the reference count for the resource currently owned by `*this`. If the reference count falls to zero, the resource is released.

Example

```
// std__memory__shared_ptr_destroy.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << "use count == " << sp1.use_count() << std::endl;

    {
        std::shared_ptr<int> sp2(sp1);
        std::cout << "*sp2 == " << *sp2 << std::endl;
        std::cout << "use count == " << sp1.use_count() << std::endl;
    }

    // check use count after sp2 is destroyed
    std::cout << "use count == " << sp1.use_count() << std::endl;

    return (0);
}
```

```
*sp1 == 5
use count == 1
*sp2 == 5
use count == 2
use count == 1
```

shared_ptr::swap

Swaps two `shared_ptr` objects.

```
void swap(shared_ptr& sp);
```

Parameters

sp

The shared pointer to swap with.

Remarks

The member function leaves the resource originally owned by `*this` subsequently owned by `sp`, and the resource originally owned by `sp` subsequently owned by `*this`. The function does not change the reference counts for the two resources and it does not throw any exceptions.

Example

```
// std__memory__shared_ptr_swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}
```

```
*sp1 == 5
*sp1 == 10
*sp1 == 5

*wp1 == 5
*wp1 == 10
*wp1 == 5
```

shared_ptr::unique

Tests if owned resource is unique.

```
bool unique() const;
```

Remarks

The member function returns **true** if no other `shared_ptr` object owns the resource that is owned by `*this`, otherwise **false**.

Example

```
// std__memory__shared_ptr_unique.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "sp1.unique() == " << std::boolalpha
              << sp1.unique() << std::endl;

    std::shared_ptr<int> sp2(sp1);
    std::cout << "sp1.unique() == " << std::boolalpha
              << sp1.unique() << std::endl;

    return (0);
}
```

```
sp1.unique() == true
sp1.unique() == false
```

shared_ptr::use_count

Counts numbers of resource owners.

```
long use_count() const;
```

Remarks

The member function returns the number of `shared_ptr` objects that own the resource that is owned by `*this`.

Example

```
// std__memory__shared_ptr_use_count.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "sp1.use_count() == "
              << sp1.use_count() << std::endl;

    std::shared_ptr<int> sp2(sp1);
    std::cout << "sp1.use_count() == "
              << sp1.use_count() << std::endl;

    return (0);
}
```

```
sp1.use_count() == 1
sp1.use_count() == 2
```

See also

[weak_ptr Class](#)

[Thread Safety in the C++ Standard Library](#)

unique_ptr Class

11/9/2018 • 6 minutes to read • [Edit Online](#)

Stores a pointer to an owned object or array. The object/array is owned by no other `unique_ptr`. The object/array is destroyed when the `unique_ptr` is destroyed.

Syntax

```

class unique_ptr {
public:
    unique_ptr();
    unique_ptr(nullptr_t Nptr);
    explicit unique_ptr(pointer Ptr);
    unique_ptr(pointer Ptr,
        typename conditional<is_reference<Del>::value, Del,
        typename add_reference<const Del>::type>::type Deleter);
    unique_ptr(pointer Ptr,
        typename remove_reference<Del>::type&& Deleter);
    unique_ptr(unique_ptr&& Right);
    template <class T2, Class Del2>
    unique_ptr(unique_ptr<T2, Del2>&& Right);
    unique_ptr(const unique_ptr& Right) = delete;
    unique_ptr& operator=(const unique_ptr& Right) = delete;
};

//Specialization for arrays:
template <class T, class D>
class unique_ptr<T[], D> {
public:
    typedef pointer;
    typedef T element_type;
    typedef D deleter_type;
    constexpr unique_ptr() noexcept;
    template <class U>
    explicit unique_ptr(U p) noexcept;
    template <class U>
    unique_ptr(U p, see below d) noexcept;
    template <class U>
    unique_ptr(U p, see below d) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }      template <class U, class E>
        unique_ptr(unique_ptr<U, E>&& u) noexcept;
    ~unique_ptr();
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E>
    unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;
    T& operator[](size_t i) const;

    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void reset(nullptr_t = nullptr) noexcept;
    template <class U>
    void reset(U p) noexcept = delete;
    void swap(unique_ptr& u) noexcept; // disable copy from lvalue unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};

```

Parameters

Right

A `unique_ptr` .

Nptr

An `rvalue` of type `std::nullptr_t` .

Ptr

A `pointer` .

Deleter

A `deleter` function that is bound to a `unique_ptr`.

Exceptions

No exceptions are generated by `unique_ptr`.

Remarks

The `unique_ptr` class supersedes `auto_ptr`, and can be used as an element of C++ Standard Library containers.

Use the [make_unique](#) helper function to efficiently create new instances of `unique_ptr`.

`unique_ptr` uniquely manages a resource. Each `unique_ptr` object stores a pointer to the object that it owns or stores a null pointer. A resource can be owned by no more than one `unique_ptr` object; when a `unique_ptr` object that owns a particular resource is destroyed, the resource is freed. A `unique_ptr` object may be moved, but not copied; for more information, see [Rvalue Reference Declarator: &&](#).

The resource is freed by calling a stored `deleter` object of type `De1` that knows how resources are allocated for a particular `unique_ptr`. The default `deleter` `default_delete<T>` assumes that the resource pointed to by `ptr` is allocated with `new`, and that it can be freed by calling `delete_ptr`. (A partial specialization `unique_ptr<T[]>` manages array objects allocated with `new[]`, and has the default `deleter` `default_delete<T[]>`, specialized to call `delete[] ptr`.)

The stored pointer to an owned resource, `stored_ptr` has type `pointer`. It is `De1::pointer` if defined, and `T *` if not. The stored `deleter` object `stored_deleter` occupies no space in the object if the `deleter` is stateless. Note that `De1` can be a reference type.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
unique_ptr	There are seven constructors for <code>unique_ptr</code> .

Typedefs

TYPE NAME	DESCRIPTION
deleter_type	A synonym for the template parameter <code>De1</code> .
element_type	A synonym for the template parameter <code>T</code> .
pointer	A synonym for <code>De1::pointer</code> if defined, otherwise <code>T *</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
get	Returns <code>stored_ptr</code> .
get_deleter	Returns a reference to <code>stored_deleter</code> .

MEMBER FUNCTION	DESCRIPTION
<code>release</code>	stores <code>pointer()</code> in <code>stored_ptr</code> and returns its previous contents.
<code>reset</code>	Releases the currently owned resource and accepts a new resource.
<code>swap</code>	Exchanges resource and <code>deleter</code> with the provided <code>unique_ptr</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator bool</code>	The operator returns a value of a type that is convertible to <code>bool</code> . The result of the conversion to <code>bool</code> is <code>true</code> when <code>get() != pointer()</code> , otherwise <code>false</code> .
<code>operator-></code>	The member function returns <code>stored_ptr</code> .
<code>operator*</code>	The member function returns <code>*stored_ptr</code> .
<code>unique_ptr operator=</code>	Assigns the value of a <code>unique_ptr</code> (or a <code>pointer-type</code>) to the current <code>unique_ptr</code> .

Requirements

Header: `<memory>`

Namespace: `std`

`deleter_type`

The type is a synonym for the template parameter `Del`.

```
typedef Del deleter_type;
```

Remarks

The type is a synonym for the template parameter `Del`.

`element_type`

The type is a synonym for the template parameter `Type`.

```
typedef Type element_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

`unique_ptr::get`

Returns `stored_ptr` .

```
pointer get() const;
```

Remarks

The member function returns `stored_ptr` .

unique_ptr::get_deleter

Returns a reference to `stored_deleter` .

```
Del& get_deleter();  
  
const Del& get_deleter() const;
```

Remarks

The member function returns a reference to `stored_deleter` .

unique_ptr operator=

Assigns the address of the provided `unique_ptr` to the current one.

```
unique_ptr& operator=(unique_ptr&& right);  
template <class U, Class Del2>  
unique_ptr& operator=(unique_ptr<Type, Del>&& right);  
unique_ptr& operator=(pointer-type);
```

Parameters

A `unique_ptr` reference used to assign the value of to the current `unique_ptr` .

Remarks

The member functions call `reset(right.release())` and move `right.stored_deleter` to `stored_deleter` , then return `*this` .

pointer

A synonym for `Del::pointer` if defined, otherwise `Type *` .

```
typedef T1 pointer;
```

Remarks

The type is a synonym for `Del::pointer` if defined, otherwise `Type *` .

unique_ptr::release

Releases ownership of the returned stored pointer to the caller and sets the stored pointer value to **nullptr**.

```
pointer release();
```

Remarks

Use `release` to take over ownership of the raw pointer stored by the `unique_ptr`. The caller is responsible for deletion of the returned pointer. The `unique_ptr` is set to the empty default-constructed state. You can assign another pointer of compatible type to the `unique_ptr` after the call to `release`.

Example

This example shows how the caller of `release` is responsible for the object returned:

```
// stl_release_unique.cpp
// Compile by using: cl /W4 /EHsc stl_release_unique.cpp
#include <iostream>
#include <memory>

struct Sample {
    int content_;
    Sample(int content) : content_(content) {
        std::cout << "Constructing Sample(" << content_ << ")" << std::endl;
    }
    ~Sample() {
        std::cout << "Deleting Sample(" << content_ << ")" << std::endl;
    }
};

void ReleaseUniquePointer() {
    // Use make_unique function when possible.
    auto up1 = std::make_unique<Sample>(3);
    auto up2 = std::make_unique<Sample>(42);

    // Take over ownership from the unique_ptr up2 by using release
    auto ptr = up2.release();
    if (up2) {
        // This statement does not execute, because up2 is empty.
        std::cout << "up2 is not empty." << std::endl;
    }
    // We are now responsible for deletion of ptr.
    delete ptr;
    // up1 deletes its stored pointer when it goes out of scope.
}

int main() {
    ReleaseUniquePointer();
}
```

Computer output:

```
Constructing Sample(3)
Constructing Sample(42)
Deleting Sample(42)
Deleting Sample(3)
```

unique_ptr::reset

Takes ownership of the pointer parameter, and then deletes the original stored pointer. If the new pointer is the same as the original stored pointer, `reset` deletes the pointer and sets the stored pointer to **nullptr**.

```
void reset(pointer ptr = pointer());
void reset(nullptr_t ptr);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the resource to take ownership of.

Remarks

Use `reset` to change the stored [pointer](#) owned by the `unique_ptr` to *ptr* and then delete the original stored pointer. If the `unique_ptr` was not empty, `reset` invokes the deleter function returned by [get_deleter](#) on the original stored pointer.

Because `reset` first stores the new pointer *ptr*, and then deletes the original stored pointer, it's possible for `reset` to immediately delete *ptr* if it is the same as the original stored pointer.

unique_ptr::swap

Exchanges pointers between two `unique_ptr` objects.

```
void swap(unique_ptr& right);
```

Parameters

right

A `unique_ptr` used to swap pointers.

Remarks

The member function swaps `stored_ptr` with `right.stored_ptr` and `stored_deleter` with `right.stored_deleter`.

unique_ptr::unique_ptr

There are seven constructors for `unique_ptr`.

```
unique_ptr();

unique_ptr(nullptr_t);
explicit unique_ptr(pointer ptr);

unique_ptr(
    Type* ptr,
    typename conditional<
        is_reference<Del>::value,
        Del,
        typename add_reference<const Del>::type::type _Deleter);

unique_ptr(pointer ptr, typename remove_reference<Del>::type&& _Deleter);
unique_ptr(unique_ptr&& right);
template <class Ty2, Class Del2>
unique_ptr(unique_ptr<Ty2, Del2>&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>ptr</i>	A pointer to the resource to be assigned to a <code>unique_ptr</code> .
<i>_Deleter</i>	A <code>deleter</code> to be assigned to a <code>unique_ptr</code> .

PARAMETER	DESCRIPTION
<i>right</i>	An rvalue reference to a <code>unique_ptr</code> from which <code>unique_ptr</code> fields are move assigned to the newly constructed <code>unique_ptr</code> .

Remarks

The first two constructors construct an object that manages no resource. The third constructor stores *ptr* in `stored_ptr`. The fourth constructor stores *ptr* in `stored_ptr` and `deleter` in `stored_deleter`.

The fifth constructor stores *ptr* in `stored_ptr` and moves `deleter` into `stored_deleter`. The sixth and seventh constructors store `right.release()` in `stored_ptr` and moves `right.get_deleter()` into `stored_deleter`.

unique_ptr ~unique_ptr

The destructor for `unique_ptr`, destroys a `unique_ptr` object.

```
~unique_ptr();
```

Remarks

The destructor calls `get_deleter()(stored_ptr)`.

See also

[<memory>](#)

weak_ptr Class

11/9/2018 • 7 minutes to read • [Edit Online](#)

Wraps a weakly linked pointer.

Syntax

```
template<class _Ty>
    class weak_ptr {
public:
    typedef Ty element_type;
    weak_ptr();
    weak_ptr(const weak_ptr&);
    template <class Other>
        weak_ptr(const weak_ptr<Other>&);
    template <class Other>
        weak_ptr(const shared_ptr<Other>&);
    weak_ptr& operator=(const weak_ptr&);
    template <class Other>
        weak_ptr& operator=(const weak_ptr<Other>&);
    template <class Other>
        weak_ptr& operator=(shared_ptr<Other>&);
    void swap(weak_ptr&);
    void reset();
    long use_count() const;
    bool expired() const;
    shared_ptr<Ty> lock() const;
};
```

Parameters

Ty

The type controlled by the weak pointer.

Remarks

The template class describes an object that points to a resource that is managed by one or more [shared_ptr Class](#) objects. The `weak_ptr` objects that point to a resource do not affect the resource's reference count. Thus, when the last `shared_ptr` object that manages that resource is destroyed the resource will be freed, even if there are `weak_ptr` objects pointing to that resource. This is essential for avoiding cycles in data structures.

A `weak_ptr` object points to a resource if it was constructed from a `shared_ptr` object that owns that resource, if it was constructed from a `weak_ptr` object that points to that resource, or if that resource was assigned to it with `operator=`. A `weak_ptr` object does not provide direct access to the resource that it points to. Code that needs to use the resource does so through a `shared_ptr` object that owns that resource, created by calling the member function `lock`. A `weak_ptr` object has expired when the resource that it points to has been freed because all of the `shared_ptr` objects that own the resource have been destroyed. Calling `lock` on a `weak_ptr` object that has expired creates an empty `shared_ptr` object.

An empty `weak_ptr` object does not point to any resources and has no control block. Its member function `lock` returns an empty `shared_ptr` object.

A cycle occurs when two or more resources controlled by `shared_ptr` objects hold mutually referencing `shared_ptr` objects. For example, a circular linked list with three elements has a head node `NO`; that node holds a

`shared_ptr` object that owns the next node, `N1`; that node holds a `shared_ptr` object that owns the next node, `N2`; that node, in turn, holds a `shared_ptr` object that owns the head node, `N0`, closing the cycle. In this situation, none of the reference counts will ever become zero, and the nodes in the cycle will not be freed. To eliminate the cycle, the last node `N2` should hold a `weak_ptr` object pointing to `N0` instead of a `shared_ptr` object. Since the `weak_ptr` object does not own `N0` it doesn't affect `N0`'s reference count, and when the program's last reference to the head node is destroyed the nodes in the list will also be destroyed.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
<code>weak_ptr</code>	Constructs a <code>weak_ptr</code> .

Methods

<code>element_type</code>	The type of the element.
<code>expired</code>	Tests if ownership has expired.
<code>lock</code>	Obtains exclusive ownership of a resource.
<code>owner_before</code>	Returns true if this <code>weak_ptr</code> is ordered before (or less than) the provided pointer.
<code>reset</code>	Releases owned resource.
<code>swap</code>	Swaps two <code>weak_ptr</code> objects.
<code>use_count</code>	Counts number of designated <code>shared_ptr</code> objects.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Replaces owned resource.

Requirements

Header: <memory>

Namespace: std

element_type

The type of the element.

```
typedef Ty element_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

```
// std_memory_weak_ptr_element_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::weak_ptr<int> wp0(sp0);
    std::weak_ptr<int>::element_type val = *wp0.lock();

    std::cout << "*wp0.lock() == " << val << std::endl;

    return (0);
}
```

```
*wp0.lock() == 5
```

expired

Tests if ownership has expired.

```
bool expired() const;
```

Remarks

The member function returns **true** if `*this` has expired, otherwise **false**.

Example

```

// std_memory_weak_ptr_expired.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int(10));
        wp = sp;
        std::cout << "wp.expired() == " << std::boolalpha
            << wp.expired() << std::endl;
        std::cout << "wp.lock() == " << *wp.lock() << std::endl;
    }

    // check expired after sp is destroyed
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;
    std::cout << "(bool)wp.lock() == " << std::boolalpha
        << (bool)wp.lock() << std::endl;

    return (0);
}

```

```

wp.expired() == false
*wp.lock() == 10
wp.expired() == true
(bool)wp.lock() == false

```

lock

Obtains exclusive ownership of a resource.

```
shared_ptr<Ty> lock() const;
```

Remarks

The member function returns an empty `shared_ptr` object if `*this` has expired; otherwise it returns a `shared_ptr` `Class<Ty>` object that owns the resource that `*this` points to.

Example

```

// std_memory_weak_ptr_lock.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int(10));
        wp = sp;
        std::cout << "wp.expired() == " << std::boolalpha
            << wp.expired() << std::endl;
        std::cout << "*wp.lock() == " << *wp.lock() << std::endl;
    }

    // check expired after sp is destroyed
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;
    std::cout << "(bool)wp.lock() == " << std::boolalpha
        << (bool)wp.lock() << std::endl;

    return (0);
}

```

```

wp.expired() == false
*wp.lock() == 10
wp.expired() == true
(bool)wp.lock() == false

```

operator=

Replaces owned resource.

```

weak_ptr& operator=(const weak_ptr& wp);

template <class Other>
weak_ptr& operator=(const weak_ptr<Other>& wp);

template <class Other>
weak_ptr& operator=(const shared_ptr<Other>& sp);

```

Parameters

Other

The type controlled by the argument shared/weak pointer.

wp

The weak pointer to copy.

sp

The shared pointer to copy.

Remarks

The operators all release the resource currently pointed to by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If an operator fails it leaves `*this` unchanged.

Example

```
// std_memory_weak_ptr_operator_as.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::weak_ptr<int> wp0(sp0);
    std::cout << "*wp0.lock() == " << *wp0.lock() << std::endl;

    std::shared_ptr<int> sp1(new int(10));
    wp0 = sp1;
    std::cout << "*wp0.lock() == " << *wp0.lock() << std::endl;

    std::weak_ptr<int> wp1;
    wp1 = wp0;
    std::cout << "*wp1.lock() == " << *wp1.lock() << std::endl;

    return (0);
}
```

```
*wp0.lock() == 5
*wp0.lock() == 10
*wp1.lock() == 10
```

owner_before

Returns **true** if this `weak_ptr` is ordered before (or less than) the provided pointer.

```
template <class Other>
bool owner_before(const shared_ptr<Other>& ptr);

template <class Other>
bool owner_before(const weak_ptr<Other>& ptr);
```

Parameters

ptr

An `lvalue` reference to either a `shared_ptr` or a `weak_ptr`.

Remarks

The template member function returns **true** if `*this` is ordered before `ptr`.

reset

Releases owned resource.

```
void reset();
```

Remarks

The member function releases the resource pointed to by `*this` and converts `*this` to an empty `weak_ptr` object.

Example

```
// std_memory_weak_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp(new int(5));
    std::weak_ptr<int> wp(sp);
    std::cout << "*wp.lock() == " << *wp.lock() << std::endl;
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;

    wp.reset();
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;

    return (0);
}
```

```
*wp.lock() == 5
wp.expired() == false
wp.expired() == true
```

swap

Swaps two `weak_ptr` objects.

```
void swap(weak_ptr& wp);
```

Parameters

wp

The weak pointer to swap with.

Remarks

The member function leaves the resource originally pointed to by `*this` subsequently pointed to by *wp*, and the resource originally pointed to by *wp* subsequently pointed to by `*this`. The function does not change the reference counts for the two resources and it does not throw any exceptions.

Example

```

// std_memory_weak_ptr_swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}

```

```

*sp1 == 5
*sp1 == 10
*sp1 == 5

*wp1 == 5
*wp1 == 10
*wp1 == 5

```

use_count

Counts number of designated `shared_ptr` objects.

```
long use_count() const;
```

Remarks

The member function returns the number of `shared_ptr` objects that own the resource pointed to by `*this`.

Example

```
// std_memory__weak_ptr_use_count.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::weak_ptr<int> wp(sp1);
    std::cout << "wp.use_count() == "
        << wp.use_count() << std::endl;

    std::shared_ptr<int> sp2(sp1);
    std::cout << "wp.use_count() == "
        << wp.use_count() << std::endl;

    return (0);
}
```

```
wp.use_count() == 1
wp.use_count() == 2
```

weak_ptr

Constructs a `weak_ptr`.

```
weak_ptr();

weak_ptr(const weak_ptr& wp);

template <class Other>
weak_ptr(const weak_ptr<Other>& wp);

template <class Other>
weak_ptr(const shared_ptr<Other>& sp);
```

Parameters

Other

The type controlled by the argument shared/weak pointer.

wp

The weak pointer to copy.

sp

The shared pointer to copy.

Remarks

The constructors each construct an object that points to the resource named by the operand sequence.

Example

```

// std__memory__weak_ptr_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp0;
    std::cout << "wp0.expired() == " << std::boolalpha
        << wp0.expired() << std::endl;

    std::shared_ptr<int> sp1(new int(5));
    std::weak_ptr<int> wp1(sp1);
    std::cout << "*wp1.lock() == "
        << *wp1.lock() << std::endl;

    std::weak_ptr<int> wp2(wp1);
    std::cout << "*wp2.lock() == "
        << *wp2.lock() << std::endl;

    return (0);
}

```

```

wp0.expired() == true
*wp1.lock() == 5
*wp2.lock() == 5

```

See also

[shared_ptr Class](#)

<mutex>

10/31/2018 • 4 minutes to read • [Edit Online](#)

Include the standard header <mutex> to define the classes `mutex`, `recursive_mutex`, `timed_mutex`, and `recursive_timed_mutex`; the templates `lock_guard` and `unique_lock`; and supporting types and functions that define mutual-exclusion code regions.

WARNING

Beginning in Visual Studio 2015, the C++ Standard Library synchronization types are based on Windows synchronization primitives and no longer use ConCRT (except when the target platform is Windows XP). The types defined in <mutex> should not be used with any ConCRT types or functions.

Syntax

```
#include <mutex>
```

Remarks

NOTE

In code that is compiled by using `/clr`, this header is blocked.

The classes `mutex` and `recursive_mutex` are *mutex types*. A mutex type has a default constructor and a destructor that does not throw exceptions. These objects have methods that provide mutual exclusion when multiple threads try to lock the same object. Specifically, a mutex type contains the methods `lock`, `try_lock`, and `unlock`:

- The `lock` method blocks the calling thread until the thread obtains ownership of the mutex. Its return value is ignored.
- The `try_lock` method tries to obtain ownership of the mutex without blocking. Its return type is convertible to **bool** and is **true** if the method obtains ownership, but is otherwise **false**.
- The `unlock` method releases the ownership of the mutex from the calling thread.

You can use mutex types as type arguments to instantiate the templates `lock_guard` and `unique_lock`. You can use objects of these types as the `Lock` argument to the wait member functions in the template [condition_variable_any](#).

A *timed mutex type* satisfies the requirements for a mutex type. In addition, it has the `try_lock_for` and `try_lock_until` methods that must be callable by using one argument and must return a type that is convertible to **bool**. A timed mutex type can define these functions by using additional arguments, provided that those additional arguments all have default values.

- The `try_lock_for` method must be callable by using one argument, `Rel_time`, whose type is an instantiation of [chrono::duration](#). The method tries to obtain ownership of the mutex, but returns within the time that is designated by `Rel_time`, regardless of success. The return value converts to **true** if the method obtains ownership; otherwise, the return value converts to **false**.

- The `try_lock_until` method must be callable by using one argument, `Abs_time`, whose type is an instantiation of `chrono::time_point`. The method tries to obtain ownership of the mutex, but returns no later than the time that is designated by `Abs_time`, regardless of success. The return value converts to **true** if the method obtains ownership; otherwise, the return value converts to **false**.

A mutex type is also known as a *lockable type*. If it does not provide the member function `try_lock`, it is a *basic lockable type*. A timed mutex type is also known as a *timed lockable type*.

Classes

NAME	DESCRIPTION
lock_guard Class	Represents a template that can be instantiated to create an object whose destructor unlocks a <code>mutex</code> .
mutex Class (C++ Standard Library)	Represents a mutex type. Use objects of this type to enforce mutual exclusion within a program.
recursive_mutex Class	Represents a mutex type. In contrast to the <code>mutex</code> class, the behavior of calling locking methods for objects that are already locked is well-defined.
recursive_timed_mutex Class	Represents a timed mutex type. Use objects of this type to enforce mutual exclusion that has time-limited blocking within a program. Unlike objects of type <code>timed_mutex</code> , the effect of calling locking methods for <code>recursive_timed_mutex</code> objects is well-defined.
timed_mutex Class	Represents a timed mutex type. Use objects of this type to enforce mutual exclusion that has time-limited blocking within a program.
unique_lock Class	Represents a template that can be instantiated to create objects that manage the locking and unlocking of a <code>mutex</code> .

Functions

NAME	DESCRIPTION
call_once	Provides a mechanism for calling a specified callable object exactly once during execution.
lock	Attempts to lock all arguments without deadlock.

Structs

NAME	DESCRIPTION
adopt_lock_t Structure	Represents a type that is used to define an <code>adopt_lock</code> .
defer_lock_t Structure	Represents a type that defines a <code>defer_lock</code> object that is used to select one of the overloaded constructors of <code>unique_lock</code> .

NAME	DESCRIPTION
once_flag Structure	Represents a struct that is used with the template function <code>call_once</code> to ensure that initialization code is called only once, even in the presence of multiple threads of execution.
try_to_lock_t Structure	Represents a struct that defines a <code>try_to_lock</code> object and is used to select one of the overloaded constructors of <code>unique_lock</code> .

Variables

NAME	DESCRIPTION
adopt_lock	Represents an object that can be passed to constructors for <code>lock_guard</code> and <code>unique_lock</code> to indicate that the mutex object that is also being passed to the constructor is locked.
defer_lock	Represents an object that can be passed to the constructor for <code>unique_lock</code> , to indicate that the constructor should not lock the mutex object that is also being passed to it.
try_to_lock	Represents an object that can be passed to the constructor for <code>unique_lock</code> to indicate that the constructor should try to unlock the <code>mutex</code> that is also being passed to it without blocking.

See also

[Header Files Reference](#)

<mutex> functions and variables

10/31/2018 • 2 minutes to read • [Edit Online](#)

adopt_lock	call_once	defer_lock
lock	try_to_lock	

adopt_lock Variable

Represents an object that can be passed to constructors for [lock_guard](#) and [unique_lock](#) to indicate that the mutex object that is also being passed to the constructor is locked.

```
const adopt_lock_t adopt_lock;
```

call_once

Provides a mechanism for calling a specified callable object exactly once during execution.

```
template <class Callable, class... Args>
void call_once(once_flag& Flag,
               Callable F&&, Args&&... A);
```

Parameters

Flag

A [once_flag](#) object that ensures that the callable object is only called once.

F

A callable object.

A

An argument list.

Remarks

If *Flag* is not valid, the function throws a [system_error](#) that has an error code of `invalid_argument`. Otherwise, the template function uses its *Flag* argument to ensure that it calls `F(A...)` successfully exactly once, regardless of how many times the template function is called. If `F(A...)` exits by throwing an exception, the call was not successful.

defer_lock Variable

Represents an object that can be passed to the constructor for [unique_lock](#). This indicates that the constructor should not lock the mutex object that's also being passed to it.

```
const defer_lock_t defer_lock;
```


lock

Attempts to lock all arguments without deadlock.

```
template <class L1, class L2, class... L3>
void lock(L1&, L2&, L3&...);
```

Remarks

The arguments to the template function must be *mutex types*, except that calls to `try_lock` might throw exceptions.

The function locks all of its arguments without deadlock by calls to `lock`, `try_lock`, and `unlock`. If a call to `lock` or `try_lock` throws an exception, the function calls `unlock` on any of the mutex objects that were successfully locked before rethrowing the exception.

try_to_lock Variable

Represents an object that can be passed to the constructor for [unique_lock](#) to indicate that the constructor should try to unlock the `mutex` that is also being passed to it without blocking.

```
const try_to_lock_t try_to_lock;
```

See also

<[mutex](#)>

adopt_lock_t Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a type that is used to define an [adopt_lock](#).

Syntax

```
struct adopt_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

[Header Files Reference](#)

[<mutex>](#)

defer_lock_t Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a type that defines a [defer_lock](#) object that is used to select one of the overloaded constructors of [unique_lock](#).

Syntax

```
struct defer_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

[Header Files Reference](#)

[<mutex>](#)

lock_guard Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a template that can be instantiated to create an object whose destructor unlocks a `mutex`.

Syntax

```
template <class Mutex>
class lock_guard;
```

Remarks

The template argument `Mutex` must name a *mutex type*.

Members

Public Typedefs

NAME	DESCRIPTION
<code>lock_guard::mutex_type</code>	Synonym for the template argument <code>Mutex</code> .

Public Constructors

NAME	DESCRIPTION
<code>lock_guard</code>	Constructs a <code>lock_guard</code> object.
<code>lock_guard::~lock_guard</code> Destructor	Unlocks the <code>mutex</code> that was passed to the constructor.

Requirements

Header: `<mutex>`

Namespace: `std`

lock_guard::lock_guard Constructor

Constructs a `lock_guard` object.

```
explicit lock_guard(mutex_type& Mtx);

lock_guard(mutex_type& Mtx, adopt_lock_t);
```

Parameters

Mtx

A *mutex type* object.

Remarks

The first constructor constructs an object of type `lock_guard` and locks *Mtx*. If *Mtx* is not a recursive mutex, it must be unlocked when this constructor is called.

The second constructor does not lock *Mtx*. *Mtx* must be locked when this constructor is called. The constructor throws no exceptions.

lock_guard::~lock_guard Destructor

Unlocks the `mutex` that was passed to the constructor.

```
~lock_guard() noexcept;
```

Remarks

If the `mutex` does not exist when the destructor runs, the behavior is undefined.

See also

[Header Files Reference](#)

[<mutex>](#)

mutex Class (C++ Standard Library)

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *mutex type*. Objects of this type can be used to enforce mutual exclusion within a program.

Syntax

```
class mutex;
```

Members

Public Constructors

NAME	DESCRIPTION
mutex	Constructs a <code>mutex</code> object.
mutex::~mutex Destructor	Releases any resources that were used by the <code>mutex</code> object.

Public Methods

NAME	DESCRIPTION
lock	Blocks the calling thread until the thread obtains ownership of the <code>mutex</code> .
native_handle	Returns the implementation-specific type that represents the mutex handle.
try_lock	Attempts to obtain ownership of the <code>mutex</code> without blocking.
unlock	Releases ownership of the <code>mutex</code> .

Requirements

Header: <mutex>

Namespace: std

mutex::lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

mutex::mutex Constructor

Constructs a `mutex` object that is not locked.

```
constexpr mutex() noexcept;
```

mutex::~mutex Destructor

Releases any resources that are used by the `mutex` object.

```
~mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

mutex::native_handle

Returns the implementation-specific type that represents the mutex handle. The mutex handle can be used in implementation-specific ways.

```
native_handle_type native_handle();
```

Return Value

`native_handle_type` is defined as a `Concurrency::critical_section *` that's cast as `void *`.

mutex::try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
bool try_lock();
```

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

mutex::unlock

Releases ownership of the `mutex`.

```
void unlock();
```

Remarks

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

[Header Files Reference](#)

<mutex>

once_flag Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a **struct** that is used with the template function [call_once](#) to ensure that initialization code is called only once, even in the presence of multiple threads of execution.

Syntax

```
struct once_flag { constexpr once_flag() noexcept; };
```

Remarks

The `once_flag` **struct** has only a default constructor.

Objects of type `once_flag` can be created, but they cannot be copied.

Requirements

Header: <mutex>

Namespace: std

See also

[Header Files Reference](#)

[<mutex>](#)

recursive_mutex Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *mutex type*. In contrast to [mutex](#), the behavior of calls to locking methods for objects that are already locked is well-defined.

Syntax

```
class recursive_mutex;
```

Members

Public Constructors

NAME	DESCRIPTION
recursive_mutex	Constructs a <code>recursive_mutex</code> object.
~recursive_mutex Destructor	Releases any resources that are used by the <code>recursive_mutex</code> object.

Public Methods

NAME	DESCRIPTION
lock	Blocks the calling thread until the thread obtains ownership of the mutex.
try_lock	Attempts to obtain ownership of the mutex without blocking.
unlock	Releases ownership of the mutex.

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the method returns immediately, and the previous lock remains in effect.

recursive_mutex

Constructs a `recursive_mutex` object that is not locked.

```
recursive_mutex();
```

~recursive_mutex

Releases any resources that are used by the object.

```
~recursive_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
bool try_lock() noexcept;
```

Return Value

true if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the function immediately returns **true**, and the previous lock remains in effect.

unlock

Releases ownership of the mutex.

```
void unlock();
```

Remarks

This method releases ownership of the `mutex` only after it is called as many times as `lock` and `try_lock` have been called successfully on the `recursive_mutex` object.

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

[Header Files Reference](#)

[<mutex>](#)

recursive_timed_mutex Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *timed mutex type*. Objects of this type are used to enforce mutual exclusion by using time-limited blocking within a program. Unlike objects of type [timed_mutex](#), the effect of calling locking methods for `recursive_timed_mutex` objects is well-defined.

Syntax

```
class recursive_timed_mutex;
```

Members

Public Constructors

NAME	DESCRIPTION
recursive_timed_mutex	Constructs a <code>recursive_timed_mutex</code> object that's not locked.
~recursive_timed_mutex Destructor	Releases any resources that are used by the <code>recursive_timed_mutex</code> object.

Public Methods

NAME	DESCRIPTION
lock	Blocks the calling thread until the thread obtains ownership of the <code>mutex</code> .
try_lock	Attempts to obtain ownership of the <code>mutex</code> without blocking.
try_lock_for	Attempts to obtain ownership of the <code>mutex</code> for a specified time interval.
try_lock_until	Attempts to obtain ownership of the <code>mutex</code> until a specified time.
unlock	Releases ownership of the <code>mutex</code> .

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the `mutex` .

```
void lock();
```

Remarks

If the calling thread already owns the `mutex` , the method returns immediately, and the previous lock remains in effect.

recursive_timed_mutex Constructor

Constructs a `recursive_timed_mutex` object that is not locked.

```
recursive_timed_mutex();
```

~recursive_timed_mutex Destructor

Releases any resources that are used by the `recursive_timed_mutex` object.

```
~recursive_timed_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
bool try_lock() noexcept;
```

Return Value

true if the method successfully obtained ownership of the `mutex` or if the calling thread already owns the `mutex` ; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex` , the function immediately returns **true**, and the previous lock remains in effect.

try_lock_for

Attempts to obtain ownership of the `mutex` without blocking.

```
template <class Rep, class Period>  
bool try_lock_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time

A `chrono::duration` object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex` .

Return Value

true if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the method immediately returns **true**, and the previous lock remains in effect.

try_lock_until

Attempts to obtain ownership of the `mutex` without blocking.

```
template <class Clock, class Duration>
bool try_lock_for(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time

A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

true if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the method immediately returns **true**, and the previous lock remains in effect.

unlock

Releases ownership of the `mutex`.

```
void unlock();
```

Remarks

This method releases ownership of the `mutex` only after it is called as many times as [lock](#), [try_lock](#), [try_lock_for](#), and [try_lock_until](#) have been called successfully on the `recursive_timed_mutex` object.

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

[Header Files Reference](#)

[<mutex>](#)

timed_mutex Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a *timed mutex type*. Objects of this type are used to enforce mutual exclusion through time-limited blocking within a program.

Syntax

```
class timed_mutex;
```

Members

Public Constructors

NAME	DESCRIPTION
timed_mutex	Constructs a <code>timed_mutex</code> object that's not locked.
timed_mutex::~timed_mutex Destructor	Releases any resources that are used by the <code>timed_mutex</code> object.

Public Methods

NAME	DESCRIPTION
lock	Blocks the calling thread until the thread obtains ownership of the <code>mutex</code> .
try_lock	Attempts to obtain ownership of the <code>mutex</code> without blocking.
try_lock_for	Attempts to obtain ownership of the <code>mutex</code> for a specified time interval.
try_lock_until	Attempts to obtain ownership of the <code>mutex</code> until a specified time.
unlock	Releases ownership of the <code>mutex</code> .

Requirements

Header: <mutex>

Namespace: std

timed_mutex::lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::timed_mutex Constructor

Constructs a `timed_mutex` object that is not locked.

```
timed_mutex();
```

timed_mutex::~~timed_mutex Destructor

Releases any resources that are used by the `mutex` object.

```
~timed_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

timed_mutex::try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
bool try_lock();
```

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::try_lock_for

Attempts to obtain ownership of the `mutex` without blocking.

```
template <class Rep, class Period>  
bool try_lock_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time

A [chrono::duration](#) object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::try_lock_until

Attempts to obtain ownership of the `mutex` without blocking.

```
template <class Clock, class Duration>
bool try_lock_for(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time

A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::unlock

Releases ownership of the `mutex`.

```
void unlock();
```

Remarks

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

[Header Files Reference](#)

[<mutex>](#)

try_to_lock_t Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a **struct** that defines a [try_to_lock](#) object. Used to select one of the overloaded constructors of [unique_lock](#).

Syntax

```
struct try_to_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

[Header Files Reference](#)

[<mutex>](#)

unique_lock Class

10/31/2018 • 5 minutes to read • [Edit Online](#)

Represents a template that can be instantiated to create objects that manage the locking and unlocking of a `mutex`.

Syntax

```
template <class Mutex>
class unique_lock;
```

Remarks

The template argument `Mutex` must name a *mutex type*.

Internally, a `unique_lock` stores a pointer to an associated `mutex` object and a **bool** that indicates whether the current thread owns the `mutex`.

Members

Public Typedefs

NAME	DESCRIPTION
<code>mutex_type</code>	Synonym for the template argument <code>Mutex</code> .

Public Constructors

NAME	DESCRIPTION
<code>unique_lock</code>	Constructs a <code>unique_lock</code> object.
<code>~unique_lock</code> Destructor	Releases any resources that are associated with the <code>unique_lock</code> object.

Public Methods

NAME	DESCRIPTION
<code>lock</code>	Blocks the calling thread until the thread obtains ownership of the associated <code>mutex</code> .
<code>mutex</code>	Retrieves the stored pointer to the associated <code>mutex</code> .
<code>owns_lock</code>	Specifies whether the calling thread owns the associated <code>mutex</code> .
<code>release</code>	Disassociates the <code>unique_lock</code> object from the associated <code>mutex</code> object.

NAME	DESCRIPTION
swap	Swaps the associated <code>mutex</code> and ownership status with that of a specified object.
try_lock	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
try_lock_for	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
try_lock_until	Attempts to obtain ownership of the associated <code>mutex</code> without blocking.
unlock	Releases ownership of the associated <code>mutex</code> .

Public Operators

NAME	DESCRIPTION
operator bool	Specifies whether the calling thread has ownership of the associated <code>mutex</code> .
operator=	Copies the stored <code>mutex</code> pointer and associated ownership status from a specified object.

Inheritance Hierarchy

unique_lock

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the associated `mutex` .

```
void lock();
```

Remarks

If the stored `mutex` pointer is NULL, this method throws a [system_error](#) that has an error code of `operation_not_permitted` .

If the calling thread already owns the associated `mutex` , this method throws a `system_error` that has an error code of `resource_deadlock_would_occur` .

Otherwise, this method calls `lock` on the associated `mutex` and sets the internal thread ownership flag to **true**.

mutex

Retrieves the stored pointer to the associated `mutex` .

```
mutex_type *mutex() const noexcept;
```

operator bool

Specifies whether the calling thread has ownership of the associated mutex.

```
explicit operator bool() noexcept
```

Return Value

true if the thread owns the mutex; otherwise **false**.

operator=

Copies the stored `mutex` pointer and associated ownership status from a specified object.

```
unique_lock& operator=(unique_lock&& Other) noexcept;
```

Parameters

Other

A `unique_lock` object.

Return Value

`*this`

Remarks

If the calling thread owns the previously associated `mutex` , before this method calls `unlock` on the `mutex` , it assigns the new values.

After the copy, this method sets *Other* to a default-constructed state.

owns_lock

Specifies whether the calling thread owns the associated `mutex` .

```
bool owns_lock() const noexcept;
```

Return Value

true if the thread owns the `mutex` ; otherwise, **false**.

release

Disassociates the `unique_lock` object from the associated `mutex` object.

```
mutex_type *release() noexcept;
```

Return Value

The previous value of the stored `mutex` pointer.

Remarks

This method sets the value of the stored `mutex` pointer to 0 and sets the internal `mutex` ownership flag to **false**.

swap

Swaps the associated `mutex` and ownership status with that of a specified object.

```
void swap(unique_lock& Other) noexcept;
```

Parameters

Other

A `unique_lock` object.

try_lock

Attempts to obtain ownership of the associated `mutex` without blocking.

```
bool try_lock() noexcept;
```

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a [system_error](#) that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

try_lock_for

Attempts to obtain ownership of the associated `mutex` without blocking.

```
template <class Rep, class Period>
bool try_lock_for(
    const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time

A [chrono::duration](#) object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a [system_error](#) that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

try_lock_until

Attempts to obtain ownership of the associated `mutex` without blocking.

```
template <class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time

A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

true if the method successfully obtains ownership of the `mutex`; otherwise, **false**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a [system_error](#) that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

unique_lock Constructor

Constructs a `unique_lock` object.

```
unique_lock() noexcept;
unique_lock(unique_lock&& Other) noexcept;
explicit unique_lock(mutex_type& Mtx);

unique_lock(mutex_type& Mtx, adopt_lock_t Adopt);

unique_lock(mutex_type& Mtx, defer_lock_t Defer) noexcept;
unique_lock(mutex_type& Mtx, try_to_lock_t Try);

template <class Rep, class Period>
unique_lock(mutex_type& Mtx,
    const chrono::duration<Rep, Period>
    Rel_time);

template <class Clock, class Duration>
unique_lock(mutex_type& Mtx,
    const chrono::time_point<Clock, Duration>
    Abs_time);

unique_lock(mutex_type& Mtx,
    const xtime* Abs_time) noexcept;
```

Parameters

Mtx

A mutex type object.

Rel_time

A [chrono::duration](#) object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Abs_time

A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex` .

Other

A `unique_lock` object.

Remarks

The first constructor constructs an object that has an associated mutex pointer value of 0.

The second constructor moves the associated mutex status from *Other*. After the move, *Other* is no longer associated with a mutex.

The remaining constructors store & *Mtx* as the stored `mutex` pointer. Ownership of the `mutex` is determined by the second argument, if it exists.

No argument	Ownership is obtained by calling the <code>lock</code> method on the associated <code>mutex</code> object.
Adopt	Ownership is assumed. <code>Mtx</code> must be locked when the constructor is called.
Defer	The calling thread is assumed not to own the <code>mutex</code> object. <code>Mtx</code> must not be locked when the constructor is called.
Try	Ownership is determined by calling <code>try_lock</code> on the associated <code>mutex</code> object. The constructor throws nothing.
Rel_time	Ownership is determined by calling <code>try_lock_for(Rel_time)</code> .
Abs_time	Ownership is determined by calling <code>try_lock_until(Abs_time)</code> .

~unique_lock Destructor

Releases any resources that are associated with the `unique_lock` object.

```
~unique_lock() noexcept;
```

Remarks

If the calling thread owns the associated `mutex` , the destructor releases ownership by calling `unlock` on the `mutex` object.

unlock

Releases ownership of the associated `mutex` .

```
void unlock();
```

Remarks

If the calling thread doesn't own the associated `mutex`, this method throws a [system_error](#) that has an error code of `operation_not_permitted`.

Otherwise, this method calls `unlock` on the associated `mutex` and sets the internal thread ownership flag to **false**.

See also

[Header Files Reference](#)

[<mutex>](#)

<new>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines several types and functions that control the allocation and freeing of storage under program control. It also defines components for reporting on storage management errors.

Syntax

```
#include <new>
```

Remarks

Some of the functions declared in this header are replaceable. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

Objects

nothrow	Provides an object to be used as an argument for the nothrow versions of new and delete .

Typedefs

TYPE NAME	DESCRIPTION
new_handler	A type that points to a function suitable for use as a new handler.

Functions

FUNCTION	DESCRIPTION
set_new_handler	Installs a user function that is called when new fails in its attempt to allocate memory.

Operators

OPERATOR	DESCRIPTION
operator delete	The function called by a delete expression to deallocate storage for individual of objects.
operator delete[]	The function called by a delete expression to deallocate storage for an array of objects.
operator new	The function called by a new expression to allocate storage for individual objects.

OPERATOR	DESCRIPTION
operator new[]	The function called by a new expression to allocate storage for an array of objects.

Classes

CLASS	DESCRIPTION
bad_alloc Class	The class describes an exception thrown to indicate that an allocation request did not succeed.
nothrow_t Class	The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<new> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

[nothrow](#)

[set_new_handler](#)

nothrow

Provides an object to be used as an argument for the **nothrow** versions of **new** and **delete**.

```
extern const std::nothrow_t nothrow;
```

Remarks

The object is used as a function argument to match the parameter type [std::nothrow_t](#).

Example

See [operator new](#) and [operator new\[\]](#) for examples of how `std::nothrow_t` is used as a function parameter.

set_new_handler

Installs a user function that is to be called when **operator new** fails in its attempt to allocate memory.

```
new_handler set_new_handler(new_handler Pnew) throw();
```

Parameters

Pnew

The `new_handler` to be installed.

Return Value

0 on the first call and the previous `new_handler` on subsequent calls.

Remarks

The function stores *Pnew* in a static [new handler](#) pointer that it maintains, then returns the value previously stored in the pointer. The new handler is used by [operator new](#)(**size_t**).

Example

```

// new_set_new_handler.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;
void __cdecl newhandler( )
{
    cout << "The new_handler is called:" << endl;
    throw bad_alloc( );
    return;
}

int main( )
{
    set_new_handler (newhandler);
    try
    {
        while ( 1 )
        {
            new int[5000000];
            cout << "Allocating 5000000 ints." << endl;
        }
    }
    catch ( exception e )
    {
        cout << e.what( ) << endl;
    }
}

```

```

Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
The new_handler is called:
bad allocation

```

See also

[<new>](#)

<new> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

`new_handler`

`new_handler`

The type points to a function suitable for use as a new handler.

```
typedef void (*new_handler)();
```

Remarks

This type of handler function is called by **operator new** or `operator new[]` when they cannot satisfy a request for additional storage.

Example

See [set_new_handler](#) for an example using `new_handler` as a return value.

See also

[<new>](#)

<new> operators

10/31/2018 • 8 minutes to read • [Edit Online](#)

operator delete	operator delete[]	operator new
operator new[]		

operator delete

The function called by a delete expression to deallocate storage for individual of objects.

```
void operator delete(void* ptr) throw();

void operator delete(void *,
    void*) throw();

void operator delete(void* ptr,
    const std::nothrow_t&) throw();
```

Parameters

ptr

The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by a delete expression to render the value of *ptr* invalid. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *ptr* that is null or that was returned by an earlier call to [operator new\(size_t\)](#).

The default behavior for a null value of *ptr* is to do nothing. Any other value of *ptr* must be a value returned earlier by a call as previously described. The default behavior for such a nonnull value of *ptr* is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a placement delete expression corresponding to a new expression of the form **new(std::size_t)**. It does nothing.

The third function is called by a placement delete expression corresponding to a new expression of the form **new(std::size_t, const std::nothrow_t&)**. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of `ptr` that is null or that was returned by an earlier call to `operator new(size_t)`. The default behavior is to evaluate **delete(ptr)**.

Example

See [operator new](#) for an example that use **operator delete**.

operator delete[]

The function called by a delete expression to deallocate storage for an array of objects.

```

void operator delete[](void* ptr) throw();

void operator delete[](void *,
    void*) throw();

void operator delete[](void* ptr,
    const std::nothrow_t&) throw();

```

Parameters

ptr

The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by an `delete[]` expression to render the value of *ptr* invalid. The function is replaceable because the program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *ptr* that is null or that was returned by an earlier call to [operator new\[\]\(**size_t**\)](#). The default behavior for a null value of *ptr* is to do nothing. Any other value of *ptr* must be a value returned earlier by a call as previously described. The default behavior for such a non-null value of *ptr* is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to [operator new\(**size_t**\)](#), or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a placement `delete[]` expression corresponding to a `new[]` expression of the form `new[](std::size_t)`. It does nothing.

The third function is called by a placement delete expression corresponding to a `new[]` expression of the form `new[](std::size_t, const std::nothrow_t&)`. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *ptr* that is null or that was returned by an earlier call to [operator new\[\]\(**size_t**\)](#). The default behavior is to evaluate `delete[](ptr)`.

Example

See [operator new\[\]](#) for examples of the use of `operator delete[]`.

operator new

The function called by a new-expression to allocate storage for individual objects.

```

void* operator new(std::size_t count) throw(bad_alloc);

void* operator new(std::size_t count,
    const std::nothrow_t&) throw();

void* operator new(std::size_t count,
    void* ptr) throw();

```

Parameters

count

The number of bytes of storage to be allocated.

ptr

The pointer to be returned.

Return Value

A pointer to the lowest byte address of the newly-allocated storage. Or *ptr*.

Remarks

The first function is called by a new expression to allocate `count` bytes of storage suitably aligned to represent any object of that size. The program can define an alternate function with this function signature that replaces the default version defined by the C++ Standard Library and so is replaceable.

The required behavior is to return a nonnull pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If `count` is zero, the value returned does not compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to `malloc (size_t)` is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise, the function calls the designated [new handler](#). If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

The required behavior of a new handler is to perform one of the following operations:

- Make more storage available for allocation and then return.
- Call either **abort** or **exit**(`int`).
- Throw an object of type **bad_alloc**.

The default behavior of a [new handler](#) is to throw an object of type `bad_alloc`. A null pointer designates the default new handler.

The order and contiguity of storage allocated by successive calls to `operator new (size_t)` is unspecified, as are the initial values stored there.

The second function is called by a placement new expression to allocate `count` bytes of storage suitably aligned to represent any object of that size. The program can define an alternate function with this function signature that replaces the default version defined by the C++ Standard Library and so is replaceable.

The default behavior is to return `operator new (count)` if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a placement **new** expression, of the form **new** (*args*) *T*. Here, *args* consists of a single object pointer. This can be useful for constructing an object at a known address. The function returns *ptr*.

To free storage allocated by **operator new**, call [operator delete](#).

For information on throwing or nonthrowing behavior of new, see [The new and delete Operators](#).

Example

```

// new_op_new.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;

class MyClass
{
public:
    MyClass( )
    {
        cout << "Construction MyClass." << this << endl;
    };

    ~MyClass( )
    {
        imember = 0; cout << "Destructing MyClass." << this << endl;
    };
    int imember;
};

int main( )
{
    // The first form of new delete
    MyClass* fPtr = new MyClass;
    delete fPtr;

    // The second form of new delete
    MyClass* fPtr2 = new( nothrow ) MyClass;
    delete fPtr2;

    // The third form of new delete
    char x[sizeof( MyClass )];
    MyClass* fPtr3 = new( &x[0] ) MyClass;
    fPtr3 -> ~MyClass();
    cout << "The address of x[0] is : " << ( void* )&x[0] << endl;
}

```

operator new[]

The allocation function called by a new expression to allocate storage for an array of objects.

```

void* operator new[](std::size_t count) throw(std::bad_alloc);

void* operator new[](std::size_t count,
    const std::nothrow_t&) throw();

void* operator new[](std::size_t count,
    void* ptr) throw();

```

Parameters

count

The number of bytes of storage to be allocated for the array object.

ptr

The pointer to be returned.

Return Value

A pointer to the lowest byte address of the newly-allocated storage. Or *ptr*.

Remarks

The first function is called by a `new[]` expression to allocate `count` bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is the same as for [operator new\(size_t\)](#). The default behavior is to return `operator new (count)`.

The second function is called by a placement `new[]` expression to allocate `count` bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The default behavior is to return **`operatornew(count)`** if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a placement `new[]` expression, of the form **`new (args) T[N]`**. Here, *args* consists of a single object pointer. The function returns `ptr`.

To free storage allocated by `operator new[]`, call [operator delete\[\]](#).

For information on throwing or nonthrowing behavior of new, see [The new and delete Operators](#).

Example

```
// new_op_alloc.cpp
// compile with: /EHsc
#include <new>
#include <iostream>

using namespace std;

class MyClass {
public:
    MyClass() {
        cout << "Construction MyClass." << this << endl;
    };

    ~MyClass() {
        imember = 0; cout << "Destructing MyClass." << this << endl;
    };
    int imember;
};

int main() {
    // The first form of new delete
    MyClass* fPtr = new MyClass[2];
    delete[ ] fPtr;

    // The second form of new delete
    char x[2 * sizeof( MyClass ) + sizeof(int)];

    MyClass* fPtr2 = new( &x[0] ) MyClass[2];
    fPtr2[1].~MyClass();
    fPtr2[0].~MyClass();
    cout << "The address of x[0] is : " << ( void* )&x[0] << endl;

    // The third form of new delete
    MyClass* fPtr3 = new( nothrow ) MyClass[2];
    delete[ ] fPtr3;
}
```

See also

[<new>](#)

bad_alloc Class

3/11/2019 • 2 minutes to read • [Edit Online](#)

The class describes an exception thrown to indicate that an allocation request did not succeed.

Syntax

```
class bad_alloc : public exception {  
    bad_alloc();  
    virtual ~bad_alloc();  
  
};
```

Remarks

The value returned by `what` is an implementation-defined C string. None of the member functions throw any exceptions.

Requirements

Header: <new>

Namespace: std

Example

```
// bad_alloc.cpp  
// compile with: /EHsc  
#include<new>  
#include<iostream>  
using namespace std;  
  
int main() {  
    char* ptr;  
    try {  
        ptr = new char[(~unsigned int((int)0)/2) - 1];  
        delete[] ptr;  
    }  
    catch( bad_alloc &ba) {  
        cout << ba.what( ) << endl;  
    }  
}
```

Sample Output

```
bad allocation
```

Requirements

Header: <new>

See also

[exception Class](#)

[Thread Safety in the C++ Standard Library](#)

nothrow_t Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

The struct is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

Syntax

```
struct std::nothrow_t {};
```

Remarks

The struct helps the compiler to select the correct version of the constructor. [nothrow](#) is a synonym for objects of type `std::nothrow_t`.

Example

See [operator new](#) and [operator new\[\]](#) for examples of how `std::nothrow_t` is used as a function parameter.

Requirements

Header: <new>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

<numeric>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines container template functions that perform algorithms for numerical processing.

Syntax

```
#include <numeric>
```

Remarks

The numeric algorithms resemble the C++ Standard Library algorithms in [<algorithm>](#), and can operate on a variety of data structures. These include standard library container classes—for example, [vector](#) and [list](#), and program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm. The algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators. The algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all pointers in the ranges must be dereferenceable and within the sequences of each range, and the last position must be reachable from the first by means of incrementation.

The algorithms extend the actions that are supported by the operations and member functions of each of the C++ Standard Library containers and enable interaction with different types of container objects at the same time.

Functions

FUNCTION	DESCRIPTION
accumulate	Computes the sum of all elements in a specified range—including some initial value—by computing successive partial sums, or computes the result of successive partial results that are obtained by using a specified binary operation instead of the sum operation.
adjacent_difference	Computes the successive differences between each element and its predecessor in an input range and outputs the results to a destination range, or computes the result of a generalized procedure where the difference operation is replaced by another specified binary operation.
inner_product	Computes the sum of the element-wise product of two ranges and adds it to a specified initial value, or computes the result of a generalized procedure where the sum and product operations are replaced by other specified binary operations.
iota	Stores a starting value, beginning with the first element and filling with successive increments of the value (<code>value++</code>) in each of the elements in the interval <code>[first, last)</code> .

FUNCTION	DESCRIPTION
partial_sum	Computes a series of sums in an input range from the first element through the <i>i</i> th element and stores the result of each sum in the <i>i</i> th element of a destination range, or computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<numeric> functions

10/31/2018 • 14 minutes to read • [Edit Online](#)

accumulate	adjacent_difference	inner_product
iota	partial_sum	

accumulate

Computes the sum of all the elements in a specified range including some initial value by computing successive partial sums or computes the result of successive partial results similarly obtained from using a specified binary operation other than the sum.

```
template <class InputIterator, class Type>
Type accumulate(InputIterator first, InputIterator last, Type val);

template <class InputIterator, class Type, class BinaryOperation>
Type accumulate(
    InputIterator first,
    InputIterator last,
    Type val,
    BinaryOperation binary_op);
```

Parameters

first

An input iterator addressing the first element in the range to be summed or combined according to a specified binary operation.

last

An input iterator addressing the last element in the range to be summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

val

An initial value to which each element is in turn added or combined with according to a specified binary operation.

binary_op

The binary operation that is to be applied to the each element in the specified range and the result of its previous applications.

Return Value

The sum of *val* and all the elements in the specified range for the first template function, or, for the second template function, the result of applying the binary operation specified, instead of the sum operation, to (*PartialResult*, **Iter*), where *PartialResult* is the result of previous applications of the operation and `Iter` is an iterator pointing to an element in the range.

Remarks

The initial value insures that there will be a well-defined result when the range is empty, in which case *val* is returned. The binary operation does not need to be associative or commutative. The result is initialized to the initial value *val* and then *result* = `binary_op (result, *Iter)` is calculated iteratively through the range, where `Iter` is an iterator pointing to successive element in the range. The range must be valid and the complexity is linear with the

size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

```
// numeric_accum.cpp
// compile with: /EHsc
#include <vector>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;

    vector<int> v1, v2(20);
    vector<int>::iterator iter1, iter2;

    int i;
    for (i = 1; i < 21; i++)
    {
        v1.push_back(i);
    }

    cout << "The original vector v1 is:\n ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")." << endl;

    // The first member function for the accumulated sum
    int total;
    total = accumulate(v1.begin(), v1.end(), 0);

    cout << "The sum of the integers from 1 to 20 is: "
        << total << "." << endl;

    // Constructing a vector of partial sums
    int j = 0, parttotal;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
    {
        parttotal = accumulate(v1.begin(), iter1 + 1, 0);
        v2[j] = parttotal;
        j++;
    }

    cout << "The vector of partial sums is:\n ( " ;
    for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
        cout << *iter2 << " ";
    cout << ")." << endl << endl;

    // The second member function for the accumulated product
    vector<int> v3, v4(10);
    vector<int>::iterator iter3, iter4;

    int s;
    for (s = 1; s < 11; s++)
    {
        v3.push_back(s);
    }

    cout << "The original vector v3 is:\n ( " ;
    for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
        cout << *iter3 << " ";
    cout << ")." << endl;

    int ptotal;
    ptotal = accumulate(v3.begin(), v3.end(), 1, multiplies<int>());
```

```

    ptotal = accumulate(v3.begin(), v3.end(), 1, multiplies<int>());

    cout << "The product of the integers from 1 to 10 is: "
          << ptotal << "." << endl;

    // Constructing a vector of partial products
    int k = 0, ppartotal;
    for (iter3 = v3.begin(); iter3 != v3.end(); iter3++) {
        ppartotal = accumulate(v3.begin(), iter3 + 1, 1, multiplies<int>());
        v4[k] = ppartotal;
        k++;
    }

    cout << "The vector of partial products is:\n ( " ;
    for (iter4 = v4.begin(); iter4 != v4.end(); iter4++)
        cout << *iter4 << " ";
    cout << ")." << endl;
}

```

```

The original vector v1 is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The sum of the integers from 1 to 20 is: 210.
The vector of partial sums is:
( 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 ).

The original vector v3 is:
( 1 2 3 4 5 6 7 8 9 10 ).
The product of the integers from 1 to 10 is: 3628800.
The vector of partial products is:
( 1 2 6 24 120 720 5040 40320 362880 3628800 ).

```

adjacent_difference

Computes the successive differences between each element and its predecessor in an input range and outputs the results to a destination range or computes the result of a generalized procedure where the difference operation is replaced by another, specified binary operation.

```

template <class InputIterator, class OutIterator>
OutputIterator adjacent_difference(
    InputIterator first,
    InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutIterator, class BinaryOperation>
OutputIterator adjacent_difference(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

```

Parameters

first

An input iterator addressing the first element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

last

An input iterator addressing the last element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

result

An output iterator addressing the first element a destination range where the series of differences or the results of the specified operation is to be stored.

binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of subtraction in the differencing procedure.

Return Value

An output iterator addressing the end of the destination range: `result` + (`last` - `first`).

Remarks

The output iterator `_result` is allowed to be the same iterator as the input iterator `* first`,* so that `adjacent_difference`s may be computed in place.

For a sequence of values a_1, a_2, a_3 , in an input range, the first template function stores successive `partial_difference`s $a_1, a_2 - a_1, a_3 - a_2$, in the destination range.

For a sequence of values a_1, a_2, a_3 , in an input range, the second template function stores successive `partial_difference`s $a_1, a_2 \text{ } binary_op \text{ } a_1, a_3 \text{ } binary_op \text{ } a_2$, in the destination range.

The binary operation `binary_op` is not required to be either associative or commutative, because the order of operations applies is completely specified.

Example

```

// numeric_adj_diff.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;

    vector<int> V1( 10 ), V2( 10 );
    vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

    list<int> L1;
    list<int>::iterator LIter1, LIterend, LIterend2;

    int t;
    for ( t = 1 ; t <= 10 ; t++ )
    {
        L1.push_back( t * t );
    }

    cout << "The input list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
        cout << *LIter1 << " ";
    cout << ")." << endl;

    // The first member function for the adjacent_differences of
    // elements in a list output to a vector
    VIterend = adjacent_difference ( L1.begin ( ) , L1.end ( ) ,
        V1.begin ( ) );

    cout << "Output vector containing adjacent_differences is:\n ( " ;
    for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
        cout << *VIter1 << " ";
    cout << ")." << endl;

    // The second member function used to compute
    // the adjacent products of the elements in a list
    VIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , V2.begin ( ) ,
        multiplies<int>( ) );

    cout << "The output vector with the adjacent products is:\n ( " ;
    for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
        cout << *VIter2 << " ";
    cout << ")." << endl;

    // Computation of adjacent_differences in place
    LIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , L1.begin ( ) );
    cout << "In place output adjacent_differences in list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != LIterend2 ; LIter1++ )
        cout << *LIter1 << " ";
    cout << ")." << endl;
}

```

inner_product

Computes the sum of the element-wise product of two ranges and adds it to a specified initial value or computes the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.

```

template <class InputIterator1, class InputIterator2, class Type>
Type inner_product(
    InputIterator1    first1,
    InputIterator1    last1,
    InputIterator2    first2,
    Type              val);

template <class InputIterator1, class InputIterator2, class Type, class BinaryOperation1, class
BinaryOperation2>
Type inner_product(
    InputIterator1    first1,
    InputIterator1    last1,
    InputIterator2    first2,
    Type              val,
    BinaryOperation1  binary_op1,
    BinaryOperation2  binary_op2);

```

Parameters

first1

An input iterator addressing the first element in the first range whose inner product or generalized inner product with the second range is to be computed.

last1

An input iterator addressing the last element in the first range whose inner product or generalized inner product with the second range is to be computed.

first2

An input iterator addressing the first element in the second range whose inner product or generalized inner product with the first range is to be computed.

val

An initial value to which the inner product or generalized inner product between the ranges is to be added.

binary_op1

The binary operation that replaces the inner product operation of sum applied to the element-wise products in the generalization of the inner product.

binary_op2

The binary operation that replaces the inner product element-wise operation of multiply in the generalization of the inner product.

Return Value

The first member function returns the sum of the element-wise products and adds to it the specified initial value. So for ranges of values a_i and b_i , it returns:

$$val + (a_1 * b_1) + (a_2 * b_2) + \dots + (a_n * b_n)$$

by iteratively replacing *val* with $val + (a_i * b_i)$.

The second member function returns:

$$val \text{ binary_op1 } (a_1 \text{ binary_op2 } b_1) \text{ binary_op1 } (a_2 \text{ binary_op2 } b_2) \text{ binary_op1 } \dots \text{ binary_op1 } (a_n \text{ binary_op2 } b_n)$$

by iteratively replacing *val* with $val \text{ binary_op1 } (a_i \text{ binary_op2 } b_i)$.

Remarks

The initial value ensures that there will be a well-defined result when the range is empty, in which case *val* is returned. The binary operations do not need to be associative or commutative. The range must be valid and the

complexity is linear with the size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

```
// numeric_inner_prod.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v1, v2(7), v3(7);
    vector<int>::iterator iter1, iter2, iter3;

    int i;
    for (i = 1; i <= 7; i++)
    {
        v1.push_back(i);
    }

    cout << "The original vector v1 is:\n ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")." << endl;

    list<int> l1, l2(7);
    list<int>::iterator lIter1, lIter2;

    int t;
    for (t = 1; t <= 7; t++)
    {
        l1.push_back(t);
    }

    cout << "The original list l1 is:\n ( " ;
    for (lIter1 = l1.begin(); lIter1 != l1.end(); lIter1++)
        cout << *lIter1 << " ";
    cout << ")." << endl;

    // The first member function for the inner product
    int inprod;
    inprod = inner_product(v1.begin(), v1.end(), l1.begin(), 0);

    cout << "The inner_product of the vector v1 and the list l1 is: "
        << inprod << "." << endl;

    // Constructing a vector of partial inner_products between v1 & l1
    int j = 0, parinprod;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++) {
        parinprod = inner_product(v1.begin(), iter1 + 1, l1.begin(), 0);
        v2[j] = parinprod;
        j++;
    }

    cout << "Vector of partial inner_products between v1 & l1 is:\n ( " ;
    for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
        cout << *iter2 << " ";
    cout << ")." << endl << endl;

    // The second member function used to compute
    // the product of the element-wise sums
    int inprod2;
```

```

inprod2 = inner_product (v1.begin(), v1.end(),
    l1.begin(), 1, multiplies<int>(), plus<int>());

cout << "The sum of the element-wise products of v1 and l1 is: "
    << inprod2 << "." << endl;

// Constructing a vector of partial sums of element-wise products
int k = 0, parinprod2;
for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
{
    parinprod2 =
        inner_product(v1.begin(), iter1 + 1, l1.begin(), 1,
            multiplies<int>(), plus<int>());
    v3[k] = parinprod2;
    k++;
}

cout << "Vector of partial sums of element-wise products is:\n ( " ;
for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
    cout << *iter3 << " ";
cout << ")." << endl << endl;
}

```

iota

Stores a starting value, beginning with the first element and filling with successive increments of that value (`value++`) in each of the elements in the interval `[first, last)` .

```

template <class ForwardIterator, class Type>
void iota(ForwardIterator first, ForwardIterator last, Type value);

```

Parameters

first

An input iterator that addresses the first element in the range to be filled.

last

An input iterator that addresses the last element in the range to be filled.

value

The starting value to store in the first element and to successively increment for subsequent elements.

Remarks

Example

The following example demonstrates some uses for the `iota` function by filling a [list](#) of integers and then filling a [vector](#) with the `list` so that the [random_shuffle](#) function can be used.


```

// compile by using: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <numeric>
#include <list>
#include <vector>
#include <iostream>

using namespace std;

int main(void)
{
    list<int> intList(10);
    vector<list<int>::iterator> intVec(intList.size());

    // Fill the list
    iota(intList.begin(), intList.end(), 0);

    // Fill the vector with the list so we can shuffle it
    iota(intVec.begin(), intVec.end(), intList.begin());

    random_shuffle(intVec.begin(), intVec.end());

    // Output results
    cout << "Contents of the integer list: " << endl;
    for (auto i: intList) {
        cout << i << ' ';
    }
    cout << endl << endl;

    cout << "Contents of the integer list, shuffled by using a vector: " << endl;
    for (auto i: intVec) {
        cout << *i << ' ';
    }
    cout << endl;
}

```

partial_sum

Computes a series of sums in an input range from the first element through the *i*th element and stores the result of each such sum in the *i*th element of a destination range or computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.

```

template <class InputIterator, class OutIt>
OutputIterator partial_sum(
    InputIterator first,
    InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutIt, class Fn2>
OutputIterator partial_sum(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

```

Parameters

first

An input iterator addressing the first element in the range to be partially summed or combined according to a specified binary operation.

last

An input iterator addressing the last element in the range to be partially summed or combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

result

An output iterator addressing the first element a destination range where the series of partial sums or the results of the specified operation is to be stored.

binary_op

The binary operation that is to be applied in the generalized operation replacing the operation of sum in the partial sum procedure.

Return Value

An output iterator addressing the end of the destination range: `result + (last - first)`,

Remarks

The output iterator *result* is allowed to be the same iterator as the input iterator *first*, so that partial sums may be computed in place.

For a sequence of values a_1, a_2, a_3 , in an input range, the first template function stores successive partial sums in the destination range, where the i th element is given by $((a_1 + a_2) + a_3) a_i$.

For a sequence of values a_1, a_2, a_3 , in an input range, the second template function stores successive partial sums in the destination range, where the i th element is given by $((a_1 \text{ `binary_op` } a_2) \text{ `binary_op` } a_3) a_i$.

The binary operation *binary_op* is not required to be either associative or commutative, because the order of operations applies is completely specified.

Example

```

// numeric_partial_sum.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> V1( 10 ), V2( 10 );
    vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

    list<int> L1;
    list<int>::iterator LIter1, LIterend;

    int t;
    for ( t = 1 ; t <= 10 ; t++ )
    {
        L1.push_back( t );
    }

    cout << "The input list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
        cout << *LIter1 << " ";
    cout << ")." << endl;

    // The first member function for the partial sums of
    // elements in a list output to a vector
    VIterend = partial_sum ( L1.begin ( ) , L1.end ( ) ,
        V1.begin ( ) );

    cout << "The output vector containing the partial sums is:\n ( " ;
    for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
        cout << *VIter1 << " ";
    cout << ")." << endl;

    // The second member function used to compute
    // the partial product of the elements in a list
    VIterend2 = partial_sum ( L1.begin ( ) , L1.end ( ) , V2.begin ( ) ,
        multiplies<int>( ) );

    cout << "The output vector with the partial products is:\n ( " ;
    for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
        cout << *VIter2 << " ";
    cout << ")." << endl;

    // Computation of partial sums in place
    LIterend = partial_sum ( L1.begin ( ) , L1.end ( ) , L1.begin ( ) );
    cout << "The in place output partial_sum list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != LIterend ; LIter1++ )
        cout << *LIter1 << " ";
    cout << ")." << endl;
}

```

See also

[<numeric>](#)

<ostream>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template class `basic_ostream`, which mediates insertions for the iostreams. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You rarely need to include it directly.)

Syntax

```
#include <ostream>
```

Typedefs

TYPE NAME	DESCRIPTION
<code>ostream</code>	Creates a type from <code>basic_ostream</code> that is specialized on char and <code>char_traits</code> specialized on char .
<code>wostream</code>	Creates a type from <code>basic_ostream</code> that is specialized on wchar_t and <code>char_traits</code> specialized on wchar_t .

Manipulators

<code>endl</code>	Terminates a line and flushes the buffer.
<code>ends</code>	Terminates a string.
<code>flush</code>	Flushes the buffer.
<code>swap</code>	Exchanges the values of the left <code>basic_ostream</code> object parameter for those of the right <code>basic_ostream</code> object parameter.

Operators

OPERATOR	DESCRIPTION
<code>operator<<</code>	Writes various types to the stream.

Classes

CLASS	DESCRIPTION
<code>basic_ostream</code>	The template class describes an object that controls insertion of elements and encoded objects into a stream buffer.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_ostream Class

3/28/2019 • 7 minutes to read • [Edit Online](#)

This template class describes an object that controls insertion of elements and encoded objects into a stream buffer with elements of type `Elem`, also known as `char_type`, whose character traits are determined by the class `Tr`, also known as `traits_type`.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ostream : virtual public basic_ios<Elem, Tr>
```

Parameters

Elem

A `char_type`.

Tr

The character `traits_type`.

Remarks

Most of the member functions that overload `operator<<` are formatted output functions. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);

if (ok)
{try
<convert and insert elements
accumulate flags in state> }
catch (...)}
{try
{setstate(badbit);

}
catch (...)}
{}
if ((exceptions() & badbit) != 0)
throw; }}
width(0);
// Except for operator<<(Elem)
setstate(state);

return (*this);
```

Two other member functions are unformatted output functions. They follow the pattern:

```

iostate state = goodbit;
const sentry ok(*this);

if (!ok)
    state |= badbit;
else
{try
{<obtain and insert elements
    accumulate flags in state> }
    catch (...)}
{try
{setstate(badbit);

}
    catch (...)}
{
    if ((exceptions() & badbit) != 0)
        throw; }}
setstate(state);

return (*this);

```

Both groups of functions call [setstate\(badbit\)](#) if they encounter a failure while inserting elements.

An object of class `basic_istream< Elem, Tr>` stores only a virtual public base object of class `basic_ios<Elem, Tr>`.

Example

See the example for [basic_ofstream Class](#) to learn more about output streams.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_ostream	Constructs a <code>basic_ostream</code> object.

Member functions

MEMBER FUNCTION	DESCRIPTION
flush	Flushes the buffer.
put	Puts a character in a stream.
seekp	Resets position in output stream.
sentry	The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.
swap	Exchanges the values of this <code>basic_ostream</code> object for those of the provided <code>basic_ostream</code> object.
tellp	Reports position in output stream.
write	Puts characters in a stream.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns the value of the provided <code>basic_ostream</code> object parameter to this object.
<code>operator<<</code>	Writes to the stream.

Requirements

Header: `<ostream>`

Namespace: `std`

`basic_ostream::basic_ostream`

Constructs a `basic_ostream` object.

```
explicit basic_ostream(  
    basic_streambuf<Elem, Tr>* strbuf,  
    bool _Isstd = false);  
  
basic_ostream(basic_ostream&& right);
```

Parameters

strbuf

An object of type `basic_streambuf`.

_Isstd

true if this is a standard stream; otherwise, **false**.

right

An rvalue reference to an object of type `basic_ostream`.

Remarks

The first constructor initializes the base class by calling `init(strbuf)`. The second constructor initializes the base class by calling `basic_ios::move(right)`.

Example

See the example for `basic_ofstream::basic_ofstream` to learn more about output streams.

`basic_ostream::flush`

Flushes the buffer.

```
basic_ostream<Elem, Tr>& flush();
```

Return Value

A reference to the `basic_ostream` object.

Remarks

If `rdbuf` is not a null pointer, the function calls `rdbuf->pubsync`. If that returns `-1`, the function calls `setstate(badbit)`. It returns ***this**.

Example

```
// basic_ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << "test";
    cout.flush();
}
```

```
test
```

basic_ostream::operator<<

Writes to the stream.

```
basic_ostream<Elem, Tr>& operator<<(
    basic_ostream<Elem, Tr>& (* Pfn)(basic_ostream<Elem, Tr>&));

basic_ostream<Elem, Tr>& operator<<(
    ios_base& (* Pfn)(ios_base&));

basic_ostream<Elem, Tr>& operator<<(
    basic_ios<Elem, Tr>& (* Pfn)(basic_ios<Elem, Tr>&));

basic_ostream<Elem, Tr>& operator<<(basic_streambuf<Elem, Tr>* strbuf);
basic_ostream<Elem, Tr>& operator<<(bool val);
basic_ostream<Elem, Tr>& operator<<(short val);
basic_ostream<Elem, Tr>& operator<<(unsigned short val);
basic_ostream<Elem, Tr>& operator<<(int __w64 val);
basic_ostream<Elem, Tr>& operator<<(unsigned int __w64 val);
basic_ostream<Elem, Tr>& operator<<(long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long __w64 val);
basic_ostream<Elem, Tr>& operator<<(long long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long long val);
basic_ostream<Elem, Tr>& operator<<(float val);
basic_ostream<Elem, Tr>& operator<<(double val);
basic_ostream<Elem, Tr>& operator<<(long double val);
basic_ostream<Elem, Tr>& operator<<(const void* val);
```

Parameters

Pfn

A function pointer.

strbuf

A pointer to a `stream_buf` object.

val

An element to write to the stream.

Return Value

A reference to the `basic_ostream` object.

Remarks

The `<ostream>` header also defines several global insertion operators. For more information, see [operator<<](#).

The first member function ensures that an expression of the form `ostr << endl` calls `endl(ostr)`, and then returns `*this`. The second and third functions ensure that other manipulators, such as `hex`, behave similarly. The remaining functions are all formatted output functions.

The function

```
basic_ostream<Elem, Tr>& operator<<(basic_streambuf<Elem, Tr>* strbuf);
```

extracts elements from *strbuf*, if *strbuf* is not a null pointer, and inserts them. Extraction stops on end of file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls `setstate(failbit)`. In any case, the function returns `*this`.

The function

```
basic_ostream<Elem, Tr>& operator<<(bool val);
```

converts `_val` to a Boolean field and inserts it by calling `use_facet<num_put<Elem, OutIt>> (getloc)`. `put(OutIt(rdbuf), *this, getloc, val)`. Here, `OutIt` is defined as `ostreambuf_iterator<Elem, Tr>`. The function returns `*this`.

The functions

```
basic_ostream<Elem, Tr>& operator<<(short val);
basic_ostream<Elem, Tr>& operator<<(unsigned short val);
basic_ostream<Elem, Tr>& operator<<(int val);
basic_ostream<Elem, Tr>& operator<<(unsigned int __w64 val);
basic_ostream<Elem, Tr>& operator<<(long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long val);
basic_ostream<Elem, Tr>& operator<<(long long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long long val);
basic_ostream<Elem, Tr>& operator<<(const void* val);
```

each convert *val* to a numeric field and insert it by calling `use_facet<num_put<Elem, OutIt>> (getloc)`. `put(OutIt(rdbuf), *this, getloc, val)`. Here, `OutIt` is defined as `ostreambuf_iterator<Elem, Tr>`. The function returns `*this`.

The functions

```
basic_ostream<Elem, Tr>& operator<<(float val);
basic_ostream<Elem, Tr>& operator<<(double val);
basic_ostream<Elem, Tr>& operator<<(long double val);
```

each convert *val* to a numeric field and insert it by calling `use_facet<num_put<Elem, OutIt>> (getloc)`. `put(OutIt(rdbuf), *this, getloc, val)`. Here, `OutIt` is defined as `ostreambuf_iterator<Elem, Tr>`. The function returns `*this`.

Example

```

// basic_ostream_op_write.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;

ios_base& hex2( ios_base& ib )
{
    ib.unsetf( ios_base::dec );
    ib.setf( ios_base::hex );
    return ib;
}

basic_ostream<char, char_traits<char> >& somefunc(basic_ostream<char, char_traits<char> > &i)
{
    if (i == cout)
    {
        i << "i is cout" << endl;
    }
    return i;
}

class CTxtStreambuf : public basic_streambuf< char, char_traits< char > >
{
public:
    CTxtStreambuf(char *_pszText)
    {
        pszText = _pszText;
        setg(pszText, pszText, pszText + strlen(pszText));
    };
    char *pszText;
};

int main()
{
    cout << somefunc;
    cout << 21 << endl;

    hex2(cout);
    cout << 21 << endl;

    CTxtStreambuf f("text in streambuf");
    cout << &f << endl;
}

```

basic_ostream::operator=

Assigns values for the provided `basic_ostream` object parameter to this object.

```
basic_ostream& operator=(basic_ostream&& right);
```

Parameters

right

An `rvalue` reference to a `basic_ostream` object.

Remarks

The member operator calls swap `(right)`.

basic_ostream::put

Puts a character in a stream.

```
basic_ostream<Elem, Tr>& put(char_type _Ch);
```

Parameters

_Ch

A character.

Return Value

A reference to the `basic_ostream` object.

Remarks

The unformatted output function inserts the element *_Ch*. It returns ***this**.

Example

```
// basic_ostream_put.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout.put( 'v' );
    cout << endl;
    wcout.put( L'l' );
}
```

```
v
l
```

basic_ostream::seekp

Reset position in output stream.

```
basic_ostream<Elem, Tr>& seekp(pos_type _Pos);

basic_ostream<Elem, Tr>& seekp(off_type _Off, ios_base::seekdir _Way);
```

Parameters

_Pos

The position in the stream.

_Off

The offset relative to *_Way*.

_Way

One of the `ios_base::seekdir` enumerations.

Return Value

A reference to the `basic_ostream` object.

Remarks

If `fail` is **false**, the first member function calls `newpos = rdbuf->pubseekpos(_Pos)`, for some `pos_type`

temporary object `newpos`. If `fail` is false, the second function calls **`newpos = rdbuf->pubseekoff(_Off, _Way)`**. In either case, if `(off_type)newpos == (off_type)(-1)` (the positioning operation fails), then the function calls **`istr.setstate(failbit)`**. Both functions return **`*this`**.

Example

```
// basic_ostream_seekp.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main()
{
    using namespace std;
    ofstream x("basic_ostream_seekp.txt");
    streamoff i = x.tellp();
    cout << i << endl;
    x << "testing";
    i = x.tellp();
    cout << i << endl;
    x.seekp(2); // Put char in third char position in file
    x << " ";

    x.seekp(2, ios::end); // Put char two after end of file
    x << "z";
}
```

```
0
7
```

basic_ostream::sentry

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.

```
class sentry { public: explicit sentry(basic_ostream<Elem, Tr>& _Ostr); operator bool() const; ~sentry(); ;
```

Remarks

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions. If **`ostr.good`** is **true** and **`ostr.tie`** is not a null pointer, the constructor calls **`ostr.tie->flush`**. The constructor then stores the value returned by `ostr.good` in `status`. A later call to `operator bool` delivers this stored value.

If `uncaught_exception` returns **false** and **`flags & unitbuf`** is nonzero, the destructor calls **`flush`**.

basic_ostream::swap

Exchanges the values of this `basic_ostream` object for the values of the provided `basic_ostream`.

```
void swap(basic_ostream& right);
```

Parameters

right

A reference to a `basic_ostream` object.

Remarks

The member function calls `basic_ios::swap` (right) to exchange the contents of this object for the contents of *right*.

basic_ostream::tellp

Report position in output stream.

```
pos_type tellp();
```

Return Value

Position in output stream.

Remarks

If `fail` is **false**, the member function returns `rdbuf->pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type` (-1).

Example

See [seekp](#) for an example using `tellp`.

basic_ostream::write

Put characters in a stream.

```
basic_ostream<Elem, Tr>& write(const char_type* str, streamsize count);
```

Parameters

count

Count of characters to put into the stream.

str

Characters to put into the stream.

Return Value

A reference to the `basic_ostream` object.

Remarks

The [unformatted output function](#) inserts the sequence of *count* elements beginning at *str*.

Example

See [streamsize](#) for an example using `write`.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<ostream> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

These are the global template functions defined in <ostream>. For member functions, see the [basic_ostream Class](#) documentation.

endl	ends	flush
swap		

endl

Terminates a line and flushes the buffer.

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& endl(
    basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem

The element type.

Ostr

An object of type **basic_ostream**.

Tr

Character traits.

Return Value

An object of type **basic_ostream**.

Remarks

The manipulator calls *Ostr.put(Ostr.widen('\n'))*, and then calls *Ostr.flush*. It returns *Ostr*.

Example

```
// ostream_endl.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << "testing" << endl;
}
```

```
testing
```

ends

Terminates a string.

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& ends(
    basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem

The element type.

Ostr

An object of type `basic_ostream`.

Tr

Character traits.

Return Value

An object of type `basic_ostream`.

Remarks

The manipulator calls `Ostr.put(Elem('\\0'))`. It returns *Ostr*.

Example

```
// ostream_ends.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << "a";
    cout << "b" << ends;
    cout << "c" << endl;
}
```

```
ab c
```

flush

Flushes the buffer.

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& flush(
    basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem

The element type.

Ostr

An object of type `basic_ostream`.

Tr

Character traits.

Return Value

An object of type `basic_ostream`.

Remarks

The manipulator calls `Ostr.flush`. It returns `Ostr`.

Example

```
// ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << "testing" << flush;
}
```

```
testing
```

swap

Exchanges the values of two `basic_ostream` objects.

```
template <class Elem, class Tr>
void swap(
    basic_ostream<Elem, Tr>& left,
    basic_ostream<Elem, Tr>& right);
```

Parameters

Elem

The element type.

Tr

Character traits.

left

An lvalue reference to a `basic_ostream` object.

right

An lvalue reference to a `basic_ostream` object.

Remarks

The template function `swap` executes `left.swap(right)`.

See also

[<ostream>](#)

<ostream> operators

10/31/2018 • 3 minutes to read • [Edit Online](#)

operator<<

operator<<

Writes various types to the stream.

```

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    const Elem* str);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    Elem _Ch);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    const char* str);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    char _Ch);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _ostr,
    char _Ch);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const signed char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    signed char _Ch);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const unsigned char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    unsigned char _Ch);

template <class _Elem, class _Tr, class T>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>&& _Ostr,
    Ty val);

```

Parameters

_Ch

A character.

_Elem

The element type.

_Ostr

A `basic_ostream` object.

str

A character string.

_Tr

Character traits.

val

The type

Return Value

The stream.

Remarks

The `basic_ostream` class also defines several insertion operators. For more information, see [basic_ostream::operator<<](#).

The template function

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _ostr,
    const Elem *str);
```

determines the length $N = \text{traits_type::length}(\text{str})$ of the sequence beginning at *str*, and inserts the sequence. If $N < _ostr.\text{width}$, then the function also inserts a repetition of $_ostr.\text{width} - N$ fill characters. The repetition precedes the sequence if $(_ostr.\text{flags} \& \text{adjustfield} \neq \text{left})$. Otherwise, the repetition follows the sequence. The function returns *_Ostr*.

The template function

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    Elem _Ch);
```

inserts the element `_Ch`. If $1 < _Ostr.\text{width}$, then the function also inserts a repetition of $_Ostr.\text{width} - 1$ fill characters. The repetition precedes the sequence if $_Ostr.\text{flags} \& \text{adjustfield} \neq \text{left}$. Otherwise, the repetition follows the sequence. It returns *_Ostr*.

The template function

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    const char *str);
```

behaves the same as

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    const Elem *str);
```

except that each element *_Ch* of the sequence beginning at *str* is converted to an object of type `Elem` by calling `_Ostr.put(_Ostr.widen(_Ch))`.

The template function

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(  
    basic_ostream<Elem, _Tr>& _Ostr,  
    char _Ch);
```

behaves the same as

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(  
    basic_ostream<Elem, _Tr>& _Ostr,  
    Elem _Ch);
```

except that *_Ch* is converted to an object of type `Elem` by calling `_Ostr.put (_Ostr.widen (_Ch))`.

The template function

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(  
    basic_ostream<char, _Tr>& _Ostr,  
    const char *str);
```

behaves the same as

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(  
    basic_ostream<Elem, _Tr>& _Ostr,  
    const Elem *str);
```

(It does not have to widen the elements before inserting them.)

The template function

```
template <class _Tr>
basic_ostream<char, Tr>& operator<<(  
    basic_ostream<char, _Tr>& _Ostr,  
    char _Ch);
```

behaves the same as

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(  
    basic_ostream<Elem, _Tr>& _Ostr,  
    Elem _Ch);
```

(It does not have to widen *_Ch* before inserting it.)

The template function

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(  
    basic_ostream<char, _Tr>& _Ostr,  
    const signed char *str);
```

returns `_Ostr << (const char *) str`.

The template function

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    signed char _Ch);
```

returns `_Ostr << (char) _Ch`.

The template function:

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const unsigned char *str);
```

returns `_Ostr << (const char *) str`.

The template function:

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    unsigned char _Ch);
```

returns `_Ostr << (char) _Ch`.

The template function:

```
template <class _Elem, class _Tr, class T>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<char, _Tr>&& _Ostr,
    T val);
```

returns `_Ostr << val` (and converts a [RValue Reference](#) to `_Ostr` to an lvalue in the process).

Example

See [flush](#) for an example using `operator<<`.

See also

[<ostream>](#)

<ostream> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

ostream	wostream
---------	----------

ostream

Creates a type from `basic_ostream` that is specialized on **char** and `char_traits` specialized on **char**.

```
typedef basic_ostream<char, char_traits<char>> ostream;
```

Remarks

The type is a synonym for template class `basic_ostream`, specialized for elements of type **char** with default character traits.

wostream

Creates a type from `basic_ostream` that is specialized on **wchar_t** and `char_traits` specialized on **wchar_t**.

```
typedef basic_ostream<wchar_t, char_traits<wchar_t>> wostream;
```

Remarks

The type is a synonym for template class `basic_ostream`, specialized for elements of type **wchar_t** with default character traits.

See also

[<ostream>](#)

<queue>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template classes `priority_queue` and `queue` and several supporting templates.

Syntax

```
#include <queue>
```

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.
<code>operator<</code>	Tests if the queue object on the left side of the operator is less than the queue object on the right side.
<code>operator<=</code>	Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.
<code>operator==</code>	Tests if the queue object on the left side of the operator is equal to the queue object on the right side.
<code>operator></code>	Tests if the queue object on the left side of the operator is greater than the queue object on the right side.
<code>operator>=</code>	Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

Classes

CLASS	DESCRIPTION
<code>queue Class</code>	A template container adaptor class that provides a restriction of functionality limiting access to the front and back elements of some underlying container type.
<code>priority_queue Class</code>	A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<queue> operators

10/31/2018 • 7 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.

```
bool operator!=(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queues are not equal; **false** if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// queue_op_ne.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queues with list base containers
    queue <int, list<int> > q1, q2, q3;

    // The following line would have caused an error because vector
    // does not support pop_front( ) and so cannot be adapted
    // by queue as a base container
    // queue <int, vector<int> > q1, q2, q3;

    q1.push( 1 );
    q2.push( 1 );
    q2.push( 2 );
    q3.push( 1 );

    if ( q1 != q2 )
        cout << "The queues q1 and q2 are not equal." << endl;
    else
        cout << "The queues q1 and q2 are equal." << endl;

    if ( q1 != q3 )
        cout << "The queues q1 and q3 are not equal." << endl;
    else
        cout << "The queues q1 and q3 are equal." << endl;
}

```

The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.

operator<

Tests if the queue object on the left side of the operator is less than the queue object on the right side.

```
bool operator<(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queue on the left side of the operator is less than and not equal to the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```
// queue_op_lt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queues with default deque base container
    queue <int> q1, q2, q3;

    q1.push( 1 );
    q1.push( 2 );
    q2.push( 5 );
    q2.push( 10 );
    q3.push( 1 );
    q3.push( 2 );

    if ( q1 < q2 )
        cout << "The queue q1 is less than the queue q2." << endl;
    else
        cout << "The queue q1 is not less than the queue q2." << endl;

    if ( q1 < q3 )
        cout << "The queue q1 is less than the queue q3." << endl;
    else
        cout << "The queue q1 is not less than the queue q3." << endl;
}
```

The queue q1 is less than the queue q2.
The queue q1 is not less than the queue q3.

operator<=

Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.

```
bool operator<=(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```

// queue_op_le.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1, q2, q3;

    q1.push( 5 );
    q1.push( 10 );
    q2.push( 1 );
    q2.push( 2 );
    q3.push( 5 );
    q3.push( 10 );

    if ( q1 <= q2 )
        cout << "The queue q1 is less than or equal to "
              << "the queue q2." << endl;
    else
        cout << "The queue q1 is greater than "
              << "the queue q2." << endl;

    if ( q1 <= q3 )
        cout << "The queue q1 is less than or equal to "
              << "the queue q3." << endl;
    else
        cout << "The queue q1 is greater than "
              << "the queue q3." << endl;
}

```

The queue q1 is greater than the queue q2.
The queue q1 is less than or equal to the queue q3.

operator==

Tests if the queue object on the left side of the operator is equal to queue object on the right side.

```
bool operator==(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queues are not equal; **false** if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// queue_op_eq.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queues with list base containers
    queue <int, list<int> > q1, q2, q3;

    // The following line would have caused an error because vector
    // does not support pop_front( ) and so cannot be adapted
    // by queue as a base container
    // queue <int, vector<int> > q1, q2, q3;

    q1.push( 1 );
    q2.push( 2 );
    q3.push( 1 );

    if ( q1 != q2 )
        cout << "The queues q1 and q2 are not equal." << endl;
    else
        cout << "The queues q1 and q2 are equal." << endl;

    if ( q1 != q3 )
        cout << "The queues q1 and q3 are not equal." << endl;
    else
        cout << "The queues q1 and q3 are equal." << endl;
}

```

```

The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.

```

operator>

Tests if the queue object on the left side of the operator is greater than the queue object on the right side.

```
bool operator>(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The greater-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

```
// queue_op_gt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1, q2, q3;

    q1.push( 1 );
    q1.push( 2 );
    q1.push( 3 );
    q2.push( 5 );
    q2.push( 10 );
    q3.push( 1 );
    q3.push( 2 );

    if ( q1 > q2 )
        cout << "The queue q1 is greater than "
              << "the queue q2." << endl;
    else
        cout << "The queue q1 is not greater than "
              << "the queue q2." << endl;

    if ( q1 > q3 )
        cout << "The queue q1 is greater than "
              << "the queue q3." << endl;
    else
        cout << "The queue q1 is not greater than "
              << "the queue q3." << endl;
}
```

The queue q1 is not greater than the queue q2.
The queue q1 is greater than the queue q3.

operator>=

Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

```
bool operator>=(const queue <Type, Container>& left, const queue <Type, Container>& right,);
```

Parameters

left

An object of type `queue`.

right

An object of type `queue`.

Return Value

true if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **false**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// queue_op_ge.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1, q2, q3;

    q1.push( 1 );
    q1.push( 2 );
    q2.push( 5 );
    q2.push( 10 );
    q3.push( 1 );
    q3.push( 2 );

    if ( q1 >= q2 )
        cout << "The queue q1 is greater than or equal to "
              << "the queue q2." << endl;
    else
        cout << "The queue q1 is less than "
              << "the queue q2." << endl;

    if ( q1 >= q3 )
        cout << "The queue q1 is greater than or equal to "
              << "the queue q3." << endl;
    else
        cout << "The queue q1 is less than "
              << "the queue q3." << endl;
}
```

The queue q1 is less than the queue q2.
The queue q1 is greater than or equal to the queue q3.

See also

[<queue>](#)

priority_queue Class

11/8/2018 • 11 minutes to read • [Edit Online](#)

A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest or of the highest priority. New elements can be added to the priority_queue and the top element of the priority_queue can be inspected or removed.

Syntax

```
template <class Type, class Container= vector <Type>, class Compare= less <typename Container ::value_type>>
class priority_queue
```

Parameters

Type

The element data type to be stored in the priority_queue.

Container

The type of the underlying container used to implement the priority_queue.

Compare

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the priority_queue. This argument is optional and the binary predicate

`less<typename Container::value_type>` is the default value.

Remarks

The elements of class `Type` stipulated in the first template parameter of a queue object are synonymous with `value_type` and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

The priority_queue orders the sequence it controls by calling a stored function object of class `Traits`. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense.

Suitable underlying container classes for priority_queue include `deque Class` and the default `vector Class` or any other sequence container that supports the operations of `front`, `push_back`, and `pop_back` and a random-access iterator. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

Adding elements to and removing elements from a `priority_queue` both have logarithmic complexity. Accessing elements in a `priority_queue` has constant complexity.

There are three types of container adaptors defined by the C++ Standard Library: `stack`, `queue`, and `priority_queue`. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The `stack Class` supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the

stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.

- The [queue Class](#) supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the queue class.
- The `priority_queue` class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Constructors

CONSTRUCTOR	DESCRIPTION
priority_queue	Constructs a <code>priority_queue</code> that is empty or that is a copy of a range of a base container object or of other <code>priority_queue</code> .

Typedefs

TYPE NAME	DESCRIPTION
container_type	A type that provides the base container to be adapted by a <code>priority_queue</code> .
size_type	An unsigned integer type that can represent the number of elements in a <code>priority_queue</code> .
value_type	A type that represents the type of object stored as an element in a <code>priority_queue</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
empty	Tests if the <code>priority_queue</code> is empty.
pop	Removes the largest element of the <code>priority_queue</code> from the top position.
push	Adds an element to the priority queue based on the priority of the element from operator<.
size	Returns the number of elements in the <code>priority_queue</code> .
top	Returns a const reference to the largest element at the top of the <code>priority_queue</code> .

Requirements

Header: <queue>

Namespace: std

priority_queue::container_type

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. The C++ Standard Library sequence container class `deque` and the default class `vector` meet the requirements to be used as the base container for a `priority_queue` object. User-defined types satisfying the requirements may also be used.

For more information on `Container`, see the Remarks section of the [priority_queue Class](#) topic.

Example

See the example for [priority_queue](#) for an example of how to declare and use `container_type`.

priority_queue::empty

Tests if a `priority_queue` is empty.

```
bool empty() const;
```

Return Value

true if the `priority_queue` is empty; **false** if the `priority_queue` is nonempty.

Example

```
// pqueue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares priority_queues with default deque base container
    priority_queue<int> q1, s2;

    q1.push( 1 );

    if ( q1.empty( ) )
        cout << "The priority_queue q1 is empty." << endl;
    else
        cout << "The priority_queue q1 is not empty." << endl;

    if ( s2.empty( ) )
        cout << "The priority_queue s2 is empty." << endl;
    else
        cout << "The priority_queue s2 is not empty." << endl;
}
```

```
The priority_queue q1 is not empty.
The priority_queue s2 is empty.
```

priority_queue::pop

Removes the largest element of the priority_queue from the top position.

```
void pop();
```

Remarks

The priority_queue must be nonempty to apply the member function. The top of the priority_queue is always occupied by the largest element in the container.

Example

```
// pqueue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    priority_queue <int> q1, s2;

    q1.push( 10 );
    q1.push( 20 );
    q1.push( 30 );

    priority_queue <int>::size_type i, iii;
    i = q1.size( );
    cout << "The priority_queue length is " << i << "." << endl;

    const int& ii = q1.top( );
    cout << "The element at the top of the priority_queue is "
         << ii << "." << endl;

    q1.pop( );

    iii = q1.size( );
    cout << "After a pop, the priority_queue length is "
         << iii << "." << endl;

    const int& iv = q1.top( );
    cout << "After a pop, the element at the top of the "
         << "priority_queue is " << iv << "." << endl;
}
```

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
After a pop, the priority_queue length is 2.
After a pop, the element at the top of the priority_queue is 20.
```

priority_queue::priority_queue

Constructs a priority_queue that is empty or that is a copy of a range of a base container object or of another priority_queue.

```

priority_queue();

explicit priority_queue(const Traits& _comp);

priority_queue(const Traits& _comp, const container_type& _Cont);

priority_queue(const priority_queue& right);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Traits& _comp);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Traits& _comp, const container_type& _Cont);

```

Parameters

_comp

The comparison function of type **const Traits** used to order the elements in the priority_queue, which defaults to compare function of the base container.

_Cont

The base container of which the constructed priority_queue is to be a copy.

right

The priority_queue of which the constructed set is to be a copy.

first

The position of the first element in the range of elements to be copied.

last

The position of the first element beyond the range of elements to be copied.

Remarks

Each of the first three constructors specifies an empty initial priority_queue, the second also specifying the type of comparison function (`comp`) to be used in establishing the order of the elements and the third explicitly specifying the `container_type` (`_Cont`) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the priority_queue *right*.

The last three constructors copy the range [*first*, *last*) of some container and use the values to initialize a priority_queue with increasing explicitness in specifying the type of comparison function of class `Traits` and `container_type`.

Example

```

// pqueue_ctor.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <deque>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function declares priority_queue

```

```

// with a default vector base container
priority_queue<int> q1;
cout << "q1 = ( ";
while ( !q1.empty( ) )
{
    cout << q1.top( ) << " ";
    q1.pop( );
}
cout << ")" << endl;

// Explicitly declares a priority_queue with nondefault
// deque base container
priority_queue<int, deque<int> > q2;
q2.push( 5 );
q2.push( 15 );
q2.push( 10 );
cout << "q2 = ( ";
while ( !q2.empty( ) )
{
    cout << q2.top( ) << " ";
    q2.pop( );
}
cout << ")" << endl;

// This method of printing out the elements of a priority_queue
// removes the elements from the priority queue, leaving it empty
cout << "After printing, q2 has " << q2.size( ) << " elements." << endl;

// The third member function declares a priority_queue
// with a vector base container and specifies that the comparison
// function greater is to be used for ordering elements
priority_queue<int, vector<int>, greater<int> > q3;
q3.push( 2 );
q3.push( 1 );
q3.push( 3 );
cout << "q3 = ( ";
while ( !q3.empty( ) )
{
    cout << q3.top( ) << " ";
    q3.pop( );
}
cout << ")" << endl;

// The fourth member function declares a priority_queue and
// initializes it with elements copied from another container:
// first, inserting elements into q1, then copying q1 elements into q4
q1.push( 100 );
q1.push( 200 );
priority_queue<int> q4( q1 );
cout << "q4 = ( ";
while ( !q4.empty( ) )
{
    cout << q4.top( ) << " ";
    q4.pop( );
}
cout << ")" << endl;

// Creates an auxiliary vector object v5 to be used to initialize q5
vector<int> v5;
vector<int>::iterator v5_Iter;
v5.push_back( 10 );
v5.push_back( 30 );
v5.push_back( 20 );
cout << "v5 = ( " ;
for ( v5_Iter = v5.begin( ) ; v5_Iter != v5.end( ) ; v5_Iter++ )
    cout << *v5_Iter << " ";
cout << ")" << endl;

// The fifth member function declares and

```

```

// initializes a priority_queue q5 by copying the
// range v5[ first, last) from vector v5
priority_queue<int> q5( v5.begin( ), v5.begin( ) + 2 );
cout << "q5 = ( ";
while ( !q5.empty( ) )
{
    cout << q5.top( ) << " ";
    q5.pop( );
}
cout << ")" << endl;

// The sixth member function declares a priority_queue q6
// with a comparison function greater and initializes q6
// by copying the range v5[ first, last) from vector v5
priority_queue<int, vector<int>, greater<int> >
    q6( v5.begin( ), v5.begin( ) + 2 );
cout << "q6 = ( ";
while ( !q6.empty( ) )
{
    cout << q6.top( ) << " ";
    q6.pop( );
}
cout << ")" << endl;
}

```

priority_queue::push

Adds an element to the priority queue based on the priority of the element from operator<.

```
void push(const Type& val);
```

Parameters

val

The element added to the top of the priority_queue.

Remarks

The top of the priority_queue is the position occupied by the largest element in the container.

Example

```
// pqueue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    priority_queue<int> q1;

    q1.push( 10 );
    q1.push( 30 );
    q1.push( 20 );

    priority_queue<int>::size_type i;
    i = q1.size( );
    cout << "The priority_queue length is " << i << "." << endl;

    const int& ii = q1.top( );
    cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;
}
```

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

priority_queue::size

Returns the number of elements in the priority_queue.

```
size_type size() const;
```

Return Value

The current length of the priority_queue.

Example

```
// pqueue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    priority_queue <int> q1, q2;
    priority_queue <int>::size_type i;

    q1.push( 1 );
    i = q1.size( );
    cout << "The priority_queue length is " << i << "." << endl;

    q1.push( 2 );
    i = q1.size( );
    cout << "The priority_queue length is now " << i << "." << endl;
}
```

```
The priority_queue length is 1.
The priority_queue length is now 2.
```

priority_queue::size_type

An unsigned integer type that can represent the number of elements in a priority_queue.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the `size_type` of the base container adapted by the priority_queue.

Example

See the example for [size](#) for an example of how to declare and use `size_type`.

priority_queue::top

Returns a const reference to the largest element at the top of the priority_queue.

```
const_reference top() const;
```

Return Value

A reference to the largest element, as determined by the `Traits` function, object of the priority_queue.

Remarks

The priority_queue must be nonempty to apply the member function.

Example

```
// pqueue_top.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    priority_queue<int> q1;

    q1.push( 10 );
    q1.push( 30 );
    q1.push( 20 );

    priority_queue<int>::size_type i;
    i = q1.size( );
    cout << "The priority_queue length is " << i << "." << endl;

    const int& ii = q1.top( );
    cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;
}
```

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```


priority_queue::value_type

A type that represents the type of object stored as an element in a priority_queue.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the `value_type` of the base container adapted by the priority_queue.

Example

```
// pqueue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares priority_queues with default deque base container
    priority_queue<int>::value_type AnInt;

    AnInt = 69;
    cout << "The value_type is AnInt = " << AnInt << endl;

    priority_queue<int> q1;
    q1.push( AnInt );
    cout << "The element at the top of the priority_queue is "
        << q1.top( ) << "." << endl;
}
```

```
The value_type is AnInt = 69
The element at the top of the priority_queue is 69.
```

See also

[Thread Safety in the C++ Standard Library](#)
[C++ Standard Library Reference](#)

queue Class

10/31/2018 • 10 minutes to read • [Edit Online](#)

A template container adaptor class that provides a restriction of functionality for some underlying container type, limiting access to the front and back elements. Elements can be added at the back or removed from the front, and elements can be inspected at either end of the queue.

Syntax

```
template <class Type, class Container = deque <Type>>
class queue
```

Parameters

Type

The element data type to be stored in the queue

Container

The type of the underlying container used to implement the queue.

Remarks

The elements of class `Type` stipulated in the first template parameter of a queue object are synonymous with [value_type](#) and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for queue include [deque](#) and [list](#), or any other sequence container that supports the operations of `front`, `back`, `push_back`, and `pop_front`. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The queue objects are equality comparable if and only if the elements of class `Type` are equality comparable, and are less-than comparable if and only if the elements of class `Type` are less-than comparable.

There are three types of container adaptors defined by the C++ Standard Library: `stack`, `queue`, and `priority_queue`. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The [stack class](#) supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
- The queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the queue class.
- The [priority_queue class](#) orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>queue</code>	Constructs a <code>queue</code> that is empty or that is a copy of a base container object.

Typedefs

TYPE NAME	DESCRIPTION
<code>container_type</code>	A type that provides the base container to be adapted by the <code>queue</code> .
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a <code>queue</code> .
<code>value_type</code>	A type that represents the type of object stored as an element in a <code>queue</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>back</code>	Returns a reference to the last and most recently added element at the back of the <code>queue</code> .
<code>empty</code>	Tests if the <code>queue</code> is empty.
<code>front</code>	Returns a reference to the first element at the front of the <code>queue</code> .
<code>pop</code>	Removes an element from the front of the <code>queue</code> .
<code>push</code>	Adds an element to the back of the <code>queue</code> .
<code>size</code>	Returns the number of elements in the <code>queue</code> .

Requirements

Header: `<queue>`

Namespace: `std`

`queue::back`

Returns a reference to the last and most recently added element at the back of the queue.

```
reference back();

const_reference back() const;
```

Return Value

The last element of the queue. If the queue is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the queue object cannot be modified. If the return value of `back` is assigned to a `reference`, the queue object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty queue. See [Checked Iterators](#) for more information.

Example

```
// queue_back.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1;

    q1.push( 10 );
    q1.push( 11 );

    int& i = q1.back( );
    const int& ii = q1.front( );

    cout << "The integer at the back of queue q1 is " << i
         << "." << endl;
    cout << "The integer at the front of queue q1 is " << ii
         << "." << endl;
}
```

queue::container_type

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. Two C++ Standard Library sequence container classes — the list class and the default deque class — meet the requirements to be used as the base container for a queue object. User-defined types satisfying the requirements may also be used.

For more information on `Container`, see the Remarks section of the [queue Class](#) topic.

Example

See the example for [queue](#) for an example of how to declare and use `container_type`.

queue::empty

Tests if a queue is empty.

```
bool empty() const;
```

Return Value

true if the queue is empty; **false** if the queue is nonempty.

Example

```
// queue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queues with default deque base container
    queue<int> q1, q2;

    q1.push( 1 );

    if ( q1.empty( ) )
        cout << "The queue q1 is empty." << endl;
    else
        cout << "The queue q1 is not empty." << endl;

    if ( q2.empty( ) )
        cout << "The queue q2 is empty." << endl;
    else
        cout << "The queue q2 is not empty." << endl;
}
```

```
The queue q1 is not empty.
The queue q2 is empty.
```

queue::front

Returns a reference to the first element at the front of the queue.

```
reference front();

const_reference front() const;
```

Return Value

The first element of the queue. If the queue is empty, the return value is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the queue object cannot be modified. If the return value of `front` is assigned to a `reference`, the queue object can be modified.

The member function returns a `reference` to the first element of the controlled sequence, which must be nonempty.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty queue. See [Checked Iterators](#) for more information.

Example

```

// queue_front.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main() {
    using namespace std;
    queue <int> q1;

    q1.push( 10 );
    q1.push( 20 );
    q1.push( 30 );

    queue <int>::size_type i;
    i = q1.size( );
    cout << "The queue length is " << i << "." << endl;

    int& ii = q1.back( );
    int& iii = q1.front( );

    cout << "The integer at the back of queue q1 is " << ii
        << "." << endl;
    cout << "The integer at the front of queue q1 is " << iii
        << "." << endl;
}

```

queue::pop

Removes an element from the front of the queue.

```
void pop();
```

Remarks

The queue must be nonempty to apply the member function. The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```

// queue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1, s2;

    q1.push( 10 );
    q1.push( 20 );
    q1.push( 30 );

    queue <int>::size_type i;
    i = q1.size( );
    cout << "The queue length is " << i << "." << endl;

    i = q1.front( );
    cout << "The element at the front of the queue is "
        << i << "." << endl;

    q1.pop( );

    i = q1.size( );
    cout << "After a pop the queue length is "
        << i << "." << endl;

    i = q1.front( );
    cout << "After a pop, the element at the front of the queue is "
        << i << "." << endl;
}

```

```

The queue length is 3.
The element at the front of the queue is 10.
After a pop the queue length is 2.
After a pop, the element at the front of the queue is 20.

```

queue::push

Adds an element to the back of the queue.

```
void push(const Type& val);
```

Parameters

val

The element added to the back of the queue.

Remarks

The back of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// queue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1;

    q1.push( 10 );
    q1.push( 20 );
    q1.push( 30 );

    queue <int>::size_type i;
    i = q1.size( );
    cout << "The queue length is " << i << "." << endl;

    i = q1.front( );
    cout << "The element at the front of the queue is "
        << i << "." << endl;
}
```

```
The queue length is 3.
The element at the front of the queue is 10.
```

queue::queue

Constructs a queue that is empty or that is a copy of a base container object.

```
queue();

explicit queue(const container_type& right);
```

Parameters

right

The **const** container of which the constructed queue is to be a copy.

Remarks

The default base container for queue is deque. You can also specify list as a base container, but you cannot specify vector, because it lacks the required `pop_front` member function.

Example


```

// queue_queue.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queue with default deque base container
    queue <char> q1;

    // Explicitly declares a queue with deque base container
    queue <char, deque<char> > q2;

    // These lines don't cause an error, even though they
    // declares a queue with a vector base container
    queue <int, vector<int> > q3;
    q3.push( 10 );
    // but the following would cause an error because vector has
    // no pop_front member function
    // q3.pop( );

    // Declares a queue with list base container
    queue <int, list<int> > q4;

    // The second member function copies elements from a container
    list<int> li1;
    li1.push_back( 1 );
    li1.push_back( 2 );
    queue <int, list<int> > q5( li1 );
    cout << "The element at the front of queue q5 is "
         << q5.front( ) << "." << endl;
    cout << "The element at the back of queue q5 is "
         << q5.back( ) << "." << endl;
}

```

```

The element at the front of queue q5 is 1.
The element at the back of queue q5 is 2.

```

queue::size

Returns the number of elements in the queue.

```

size_type size() const;

```

Return Value

The current length of the queue.

Example

```
// queue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;
    queue <int> q1, q2;
    queue <int>::size_type i;

    q1.push( 1 );
    i = q1.size( );
    cout << "The queue length is " << i << "." << endl;

    q1.push( 2 );
    i = q1.size( );
    cout << "The queue length is now " << i << "." << endl;
}
```

```
The queue length is 1.
The queue length is now 2.
```

queue::size_type

An unsigned integer type that can represent the number of elements in a queue.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the `size_type` of the base container adapted by the queue.

Example

See the example for [queue::front](#) for an example of how to declare and use `size_type`.

queue::value_type

A type that represents the type of object stored as an element in a queue.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the `value_type` of the base container adapted by the queue.

Example

```
// queue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares queues with default deque base container
    queue<int>::value_type AnInt;

    AnInt = 69;
    cout << "The value_type is AnInt = " << AnInt << endl;

    queue<int> q1;
    q1.push(AnInt);
    cout << "The element at the front of the queue is "
        << q1.front( ) << "." << endl;
}
```

```
The value_type is AnInt = 69
The element at the front of the queue is 69.
```

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<random>

10/31/2018 • 15 minutes to read • [Edit Online](#)

Defines facilities for random number generation, allowing creation of uniformly distributed random numbers.

Syntax

```
#include <random>
```

Summary

A *random number generator* is an object that produces a sequence of pseudo-random values. A generator that produces values that are uniformly distributed in a specified range is a *Uniform Random Number Generator* (URNG). A template class designed to function as a URNG is referred to as an *engine* if that class has certain common traits, which are discussed later in this article. A URNG can be—and usually is—combined with a *distribution* by passing the URNG as an argument to the distribution's `operator()` to produce values that are distributed in a manner that is defined by the distribution.

These links jump to the major sections of this article:

- [Examples](#)
- [Categorized Listing](#)
- [Engines and Distributions](#)
- [Remarks](#)

Quick Tips

Here are some tips to keep in mind when using <random>:

- For most purposes, URNGs produce raw bits that must be shaped by distributions. (A notable exception to this is `std::shuffle()` because it uses a URNG directly.)
- A single instantiation of a URNG or distribution cannot safely be called concurrently because running a URNG or distribution is a modifying operation. For more information, see [Thread Safety in the C++ Standard Library](#).
- [Predefined typedefs](#) of several engines are provided; this is the preferred way to create a URNG if an engine is being used.
- The most useful pairing for most applications is the `mt19937` engine with `uniform_int_distribution`, as shown in the [code example](#) later in this article.

There are many options to choose from in the <random> header, and any of them is preferable to the outdated C Runtime function `rand()`. For information about what's wrong with `rand()` and how <random> addresses these shortcomings, see [this video](#).

Examples

The following code example shows how to generate some random numbers in this case five of them using a generator created with non-deterministic seed.

```
#include <random>
#include <iostream>

using namespace std;

int main()
{
    random_device rd;    // non-deterministic generator
    mt19937 gen(rd());    // to seed mersenne twister.
                        // replace the call to rd() with a
                        // constant value to get repeatable
                        // results.

    for (int i = 0; i < 5; ++i) {
        cout << gen() << " "; // print the raw output of the generator.
    }
    cout << endl;
}
```

```
2430338871 3531691818 2723770500 3252414483 3632920437
```

While these are high quality random numbers and different every time this program is run, they are not necessarily in a useful range. To control the range, use a uniform distribution as shown in the following code:

```
#include <random>
#include <iostream>

using namespace std;

int main()
{
    random_device rd;    // non-deterministic generator
    mt19937 gen(rd());    // to seed mersenne twister.
    uniform_int_distribution<> dist(1,6); // distribute results between 1 and 6 inclusive.

    for (int i = 0; i < 5; ++i) {
        cout << dist(gen) << " "; // pass the generator to the distribution.
    }
    cout << endl;
}
```

```
5 1 6 1 2
```

The next code example shows a more realistic set of use cases with uniformly distributed random number generators shuffling the contents of a vector and an array.

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <array>
#include <iostream>
#include <random>
#include <string>
#include <vector>
#include <functional> // ref()
```

```

using namespace std;

template <typename C> void print(const C& c) {
    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

template <class URNG>
void test(URNG& urng) {

    // Uniform distribution used with a vector
    // Distribution is [-5, 5] inclusive
    uniform_int_distribution<int> dist(-5, 5);
    vector<int> v;

    for (int i = 0; i < 20; ++i) {
        v.push_back(dist(urng));
    }

    cout << "Randomized vector: ";
    print(v);

    // Shuffle an array
    // (Notice that shuffle() takes a URNG, not a distribution)
    array<string, 26> arr = { { "H", "He", "Li", "Be", "B", "C", "N", "O", "F",
        "Ne", "Na", "Mg", "Al", "Si", "P", "S", "Cl", "Ar", "K", "Ca", "Sc",
        "Ti", "V", "Cr", "Mn", "Fe" } };

    shuffle(arr.begin(), arr.end(), urng);

    cout << "Randomized array: ";
    print(arr);
    cout << "--" << endl;
}

int main()
{
    // First run: non-seedable, non-deterministic URNG random_device
    // Slower but crypto-secure and non-repeatable.
    random_device rd;
    cout << "Using random_device URNG:" << endl;
    test(rd);

    // Second run: simple integer seed, repeatable results
    cout << "Using constant-seed mersenne twister URNG:" << endl;
    mt19937 engine1(12345);
    test(engine1);

    // Third run: random_device as a seed, different each run
    // (Desirable for most purposes)
    cout << "Using non-deterministic-seed mersenne twister URNG:" << endl;
    mt19937 engine2(rd());
    test(engine2);

    // Fourth run: "warm-up" sequence as a seed, different each run
    // (Advanced uses, allows more than 32 bits of randomness)
    cout << "Using non-deterministic-seed \"warm-up\" sequence mersenne twister URNG:" << endl;
    array<unsigned int, mt19937::state_size> seed_data;
    generate_n(seed_data.begin(), seed_data.size(), ref(rd));
    seed_seq seq(begin(seed_data), end(seed_data));
    mt19937 engine3(seq);
    test(engine3);
}

```

```

Using random_device URNG:
Randomized vector: 5 -4 2 3 0 5 -2 0 4 2 -1 2 -4 -3 1 4 4 1 2 -2
Randomized array: O Li V K C Ti N Mg Ne Sc Cl B Cr Mn Ca Al F P Na Be Si Ar Fe S He H
--
Using constant-seed mersenne twister URNG:
Randomized vector: 3 -1 -5 0 0 5 3 -4 -3 -4 1 -3 0 -3 -2 -4 5 1 -1 -1
Randomized array: Al O Ne Si Na Be C N Cr Mn H V F Sc Mg Fe K Ca S Ti B P Ar Cl Li He
--
Using non-deterministic-seed mersenne twister URNG:
Randomized vector: 5 -4 0 2 1 -2 4 4 -4 0 0 4 -5 4 -5 -1 -3 0 0 3
Randomized array: Si Fe Al Ar Na P B Sc H F Mg Li C Ti He N Mn Be O Ca Cr V K Ne Cl S
--
Using non-deterministic-seed "warm-up" sequence mersenne twister URNG:
Randomized vector: -1 3 -2 4 1 3 0 -5 5 -5 0 0 5 0 -3 3 -4 2 5 0
Randomized array: Si C Sc H Na O S Cr K Li Al Ti Cl B Mn He Fe Ne Be Ar V P Ca N Mg F
--

```

This code demonstrates two different randomizations—randomize a vector of integers and shuffle an array of indexed data—with a test template function. The first call to the test function uses the cryptographically secure, non-deterministic, not-seedable, non-repeatable URNG `random_device`. The second test run uses `mersenne_twister_engine` as URNG, with a deterministic 32-bit constant seed, which means the results are repeatable. The third test run seeds `mersenne_twister_engine` with a 32-bit non-deterministic result from `random_device`. The fourth test run expands on this by using a [seed sequence](#) filled with `random_device` results, which effectively gives more than 32-bit non-deterministic randomness (but still not crypto-secure). For more information, read on.

Categorized Listing

Uniform Random Number Generators

URNGs are often described in terms of these properties:

1. **Period length:** How many iterations it takes to repeat the sequence of numbers generated. The longer the better.
2. **Performance:** How quickly numbers can be generated and how much memory it takes. The smaller the better.
3. **Quality:** How close to true random numbers the generated sequence is. This is often called "*randomness*".

The following sections list the uniform random number generators (URNGs) provided in the `<random>` header.

Non-Deterministic Generator

random_device Class	Generates a non-deterministic, cryptographically secure random sequence by using an external device. Usually used to seed an engine. Low performance, very high quality. For more information, see Remarks .
-------------------------------------	--

Engine Typedefs with Predefined Parameters

For instantiating engines and engine adaptors. For more information, see [Engines and Distributions](#).

- `default_random_engine` The default engine.

```
typedef mt19937 default_random_engine;
```

- `knuth_b` Knuth engine.

```
typedef shuffle_order_engine<minstd_rand0, 256> knuth_b;
```

- `minstd_rand0` 1988 minimal standard engine (Lewis, Goodman, and Miller, 1969).

```
typedef linear_congruential_engine<unsigned int, 16807, 0, 2147483647> minstd_rand0;
```

- `minstd_rand` Updated minimal standard engine `minstd_rand0` (Park, Miller, and Stockmeyer, 1993).

```
typedef linear_congruential_engine<unsigned int, 48271, 0, 2147483647> minstd_rand;
```

- `mt19937` 32-bit Mersenne twister engine (Matsumoto and Nishimura, 1998).

```
typedef mersenne_twister_engine<
    unsigned int, 32, 624, 397,
    31, 0x9908b0df,
    11, 0xffffffff,
    7, 0x9d2c5680,
    15, 0xefc60000,
    18, 1812433253> mt19937;
```

- `mt19937_64` 64-bit Mersenne twister engine (Matsumoto and Nishimura, 2000).

```
typedef mersenne_twister_engine<
    unsigned long long, 64, 312, 156,
    31, 0xb5026f5aa96619e9ULL,
    29, 0x5555555555555555ULL,
    17, 0x71d67ffeda60000ULL,
    37, 0xfff7eee000000000ULL,
    43, 6364136223846793005ULL> mt19937_64;
```

- `ranlux24` 24-bit RANLUX engine (Martin Lüscher and Fred James, 1994).

```
typedef discard_block_engine<ranlux24_base, 223, 23> ranlux24;
```

- `ranlux24_base` Used as a base for `ranlux24`.

```
typedef subtract_with_carry_engine<unsigned int, 24, 10, 24> ranlux24_base;
```

- `ranlux48` 48-bit RANLUX engine (Martin Lüscher and Fred James, 1994).

```
typedef discard_block_engine<ranlux48_base, 389, 11> ranlux48;
```

- `ranlux48_base` Used as a base for `ranlux48`.

```
typedef subtract_with_carry_engine<unsigned long long, 48, 5, 12> ranlux48_base;
```

Engine Templates

Engine templates are used as standalone URNGs or as base engines passed to [engine adaptors](#).

Usually these are instantiated with a [predefined engine typedef](#) and passed to a [distribution](#). For more information, see the [Engines and Distributions](#) section.

linear_congruential_engine Class	Generates a random sequence by using the linear congruential algorithm. Most simplistic and lowest quality.
mersenne_twister_engine Class	Generates a random sequence by using the Mersenne twister algorithm. Most complex, and is highest quality except for the random_device class. Very fast performance.
subtract_with_carry_engine Class	Generates a random sequence by using the subtract-with-carry algorithm. An improvement on <code>linear_congruential_engine</code> , but much lower quality and performance than <code>mersenne_twister_engine</code> .

Engine Adaptor Templates

Engine adaptors are templates that adapt other (base) engines. Usually these are instantiated with a [predefined engine typedef](#) and passed to a [distribution](#). For more information, see the [Engines and Distributions](#) section.

discard_block_engine Class	Generates a random sequence by discarding values returned by its base engine.
independent_bits_engine Class	Generates a random sequence with a specified number of bits by repacking bits from the values returned by its base engine.
shuffle_order_engine Class	Generates a random sequence by reordering the values returned from its base engine.

[[Engine Templates](#)]

Random Number Distributions

The following sections list the distributions provided in the `<random>` header. Distributions are a post-processing mechanism, usually using URNG output as input and distributing the output by a defined statistical probability density function. For more information, see the [Engines and Distributions](#) section.

Uniform Distributions

uniform_int_distribution Class	Produces a uniform integer value distribution across a range in the closed interval [a, b] (inclusive-inclusive).
uniform_real_distribution Class	Produces a uniform real (floating-point) value distribution across a range in the half-open interval [a, b) (inclusive-exclusive).
generate_canonical	Produces an even distribution of real (floating point) values of a given precision across [0, 1) (inclusive-exclusive).

[Random Number Distributions]

Bernoulli Distributions

bernoulli_distribution Class	Produces a Bernoulli distribution of bool values.
binomial_distribution Class	Produces a binomial distribution of integer values.
geometric_distribution Class	Produces a geometric distribution of integer values.
negative_binomial_distribution Class	Produces a negative binomial distribution of integer values.

[Random Number Distributions]

Normal Distributions

cauchy_distribution Class	Produces a Cauchy distribution of real (floating point) values.
chi_squared_distribution Class	Produces a chi-squared distribution of real (floating point) values.
fisher_f_distribution Class	Produces an F-distribution (also known as Snedecor's F distribution or the Fisher-Snedecor distribution) of real (floating point) values.
lognormal_distribution Class	Produces a log-normal distribution of real (floating point) values.
normal_distribution Class	Produces a normal (Gaussian) distribution of real (floating point) values.
student_t_distribution Class	Produces a Student's <i>t</i> -distribution of real (floating point) values.

[Random Number Distributions]

Poisson Distributions

exponential_distribution Class	Produces an exponential distribution of real (floating point) values.
extreme_value_distribution Class	Produces an extreme value distribution of real (floating point) values.
gamma_distribution Class	Produces a gamma distribution of real (floating point) values.
poisson_distribution Class	Produces a Poisson distribution of integer values.
weibull_distribution Class	Produces a Weibull distribution of real (floating point) values.

[\[Random Number Distributions\]](#)

Sampling Distributions

discrete_distribution Class	Produces a discrete integer distribution.
piecewise_constant_distribution Class	Produces a piecewise constant distribution of real (floating point) values.
piecewise_linear_distribution Class	Produces a piecewise linear distribution of real (floating point) values.

[\[Random Number Distributions\]](#)

Utility Functions

This section lists the general utility functions provided in the `<random>` header.

seed_seq Class	Generates a non-biased scrambled seed sequence. Used to avoid replication of random variate streams. Useful when many URNGs are instantiated from engines.

Operators

This section lists the operators provided in the `<random>` header.

<code>operator==</code>	Tests whether the URNG on the left side of the operator is equal to the engine on the right side.
<code>operator!=</code>	Tests whether the URNG on the left side of the operator is not equal to the engine on the right side.
<code>operator<<</code>	Writes state information to a stream.
<code>operator>></code>	Extracts state information from a stream.

Engines and Distributions

Refer to the following sections for information about each of these template class categories defined in `<random>`. Both of these template class categories take a type as an argument and use shared template parameter names to describe the properties of the type that are permitted as an actual argument type, as follows:

- `IntType` indicates a **short**, **int**, **long**, **long long**, **unsigned short**, **unsigned int**, **unsigned long**, or **unsigned long long**.
- `UIntType` indicates **unsigned short**, **unsigned int**, **unsigned long**, or **unsigned long long**.
- `RealType` indicates a **float**, **double**, or **long double**.

Engines

[Engine Templates](#) and [Engine Adaptor Templates](#) are templates whose parameters customize the generator created.

An *engine* is a class or template class whose instances (generators) act as a source of random numbers uniformly distributed between a minimum and maximum value. An *engine adaptor* delivers a sequence of values that have different randomness properties by taking values produced by some other random number engine and applying an algorithm of some kind to those values.

Every engine and engine adaptor has the following members:

- `typedef numeric_type result_type` is the type that is returned by the generator's `operator()`. The `numeric_type` is passed as a template parameter on instantiation.
- `result_type operator()` returns values that are uniformly distributed between `min()` and `max()`.
- `result_type min()` returns the minimum value that is returned by the generator's `operator()`. Engine adaptors use the base engine's `min()` result.
- `result_type max()` returns the maximum value that is returned by the generator's `operator()`. When `result_type` is an integral (integer-valued) type, `max()` is the maximum value that can actually be returned (inclusive); when `result_type` is a floating-point (real-valued) type, `max()` is the smallest value greater than all values that can be returned (non-inclusive). Engine adaptors use the base engine's `max()` result.
- `void seed(result_type s)` seeds the generator with seed value `s`. For engines, the signature is `void seed(result_type s = default_seed)` for default parameter support (engine adaptors define a separate `void seed()`, see next subsection).
- `template <class Seq> void seed(Seq& q)` seeds the generator by using a `seed_seq` `Seq`.
- An explicit constructor with argument `result_type x` that creates a generator seeded as if by calling `seed(x)`.
- An explicit constructor with argument `seed_seq& seq` that creates a generator seeded as if by calling `seed(seq)`.
- `void discard(unsigned long long count)` effectively calls `operator()` `count` times and discards each value.

Engine adaptors additionally support these members (`Engine` is the first template parameter of an engine adaptor, designating the base engine's type):

- A default constructor to initialize the generator as if from the base engine's default constructor.
- An explicit constructor with argument `const Engine& eng`. This is to support copy construction using the base engine.
- An explicit constructor with argument `Engine&& eng`. This is to support move construction using the base engine.
- `void seed()` that initializes the generator with the base engine's default seed value.
- `const Engine& base()` property function that returns the base engine that was used to construct the generator.

Every engine maintains a *state* that determines the sequence of values that will be generated by subsequent calls to `operator()`. The states of two generators instantiated from engines of the same type can be compared by using `operator==` and `operator!=`. If the two states compare as equal, they will generate the same sequence of values. The state of an object can be saved to a stream as a sequence of 32-bit unsigned values by using the `operator<<` of the generator. The state is not

changed by saving it. A saved state can be read into generator instantiated from an engine of the same type by using `operator>>`.

Distributions

A [Random Number Distributions](#) is a class or template class whose instances transform a stream of uniformly distributed random numbers obtained from an engine into a stream of random numbers that have a particular distribution. Every distribution has the following members:

- `typedef numeric_type result_type` is the type that is returned by the distribution's `operator()`. The `numeric_type` is passed as a template parameter on instantiation.
- `template <class URNG> result_type operator()(URNG& gen)` returns values that are distributed according to the distribution's definition, by using `gen` as a source of uniformly distributed random values and the stored *parameters of the distribution*.
- `template <class URNG> result_type operator()(URNG& gen, param_type p)` returns values distributed in accordance with the distribution's definition, using `gen` as a source of uniformly distributed random values and the parameters structure `p`.
- `typedef unspecified_type param_type` is the package of parameters optionally passed to `operator()` and is used in place of the stored parameters to generate its return value.
- A `const param&` constructor initializes the stored parameters from its argument.
- `param_type param() const` gets the stored parameters.
- `void param(const param_type&)` sets the stored parameters from its argument.
- `result_type min()` returns the minimum value that is returned by the distribution's `operator()`.
- `result_type max()` returns the maximum value that is returned by the distribution's `operator()`. When `result_type` is an integral (integer-valued) type, `max()` is the maximum value that can actually be returned (inclusive); when `result_type` is a floating-point (real-valued) type, `max()` is the smallest value greater than all values that can be returned (non-inclusive).
- `void reset()` discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

A parameter structure is an object that stores all of the parameters needed for a distribution. It contains:

- `typedef distribution_type distribution_type`, which is the type of its distribution.
- One or more constructors that take the same parameter lists as the distribution constructors take.
- The same parameter-access functions as the distribution.
- Equality and inequality comparison operators.

For more information, see the reference subtopics below this one, linked previously in this article.

Remarks

There are two highly useful URNGs in Visual Studio—`mt19937` and `random_device`—as shown in this comparison table:

URNG	FAST	CRYPTO-SECURE	SEEDABLE	DETERMINISTIC
<code>mt19937</code>	Yes	No	Yes	Yes*
<code>random_device</code>	No	Yes	No	No

* When provided with a known seed.

Although the ISO C++ Standard does not require `random_device` to be cryptographically secure, in Visual Studio it is implemented to be cryptographically secure. (The term "cryptographically secure" does not imply guarantees, but refers to a minimum level of entropy—and therefore, the level of predictability—a given randomization algorithm provides. For more information, see the Wikipedia article [Cryptographically secure pseudorandom number generator](#).) Because the ISO C++ Standard does not require this, other platforms may implement `random_device` as a simple pseudo-random number generator (not cryptographically secure) and may only be suitable as a seed source for another generator. Check the documentation for those platforms when using `random_device` in cross-platform code.

By definition, `random_device` results are not reproducible, and a side-effect is that it may run significantly slower than other URNGs. Most applications that are not required to be cryptographically secure use `mt19937` or a similar engine, although you may want to seed it with a call to `random_device`, as shown in the [code example](#).

See also

[Header Files Reference](#)

<random> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

generate_canonical

Returns a floating-point value from a random sequence.

NOTE

The ISO C++ Standard states that this function should return values in the range [0, 1). Visual Studio is not yet compliant with this constraint. As a workaround to generate values in this range, use [uniform_real_distribution](#).

```
template <class RealType, size_t Bits, class Generator>
RealType generate_canonical(Generator& Gen);
```

Parameters

RealType

The floating point integral type. For possible types, see [<random>](#).

Bits

The random number generator.

Gen

The random number generator.

Remarks

The template function calls `operator()` of *Gen* repeatedly and packs the returned values into a floating-point value `x` of type *RealType* until it has gathered the specified number of mantissa bits in `x`. The specified number is the smaller of *Bits* (which must be nonzero) and the full number of mantissa bits in *RealType*. The first call supplies the lowest-order bits. The function returns `x`.

See also

[<random>](#)

bernoulli_distribution Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Generates a Bernoulli distribution.

Syntax

```
class bernoulli_distribution
{
public:
    // types
    typedef bool result_type;
    struct param_type;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The class describes a distribution that produces values of type **bool**, distributed according to the Bernoulli distribution discrete probability function. The following table links to articles about individual members.

bernoulli_distribution	<code>bernoulli_distribution::p</code>	<code>bernoulli_distribution::param</code>
<code>bernoulli_distribution::operator()</code>		param_type

The property member `p()` returns the currently stored distribution parameter value `p`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the Bernoulli distribution discrete probability function, see the Wolfram MathWorld article [Bernoulli Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::bernoulli_distribution distr(p);

    std::cout << "p == " << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<bool, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::boolalpha << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 0.5;
    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a double value for p distribution (where 0.0 <= p <= 1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .45
Enter an integer value for a sample count: 100
p == 0.45
Histogram for 100 samples:
false ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
true  ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Requirements

Header: <random>

Namespace: std

bernoulli_distribution::bernoulli_distribution

Constructs the distribution.

```
explicit bernoulli_distribution(double p = 0.5);
explicit bernoulli_distribution(const param_type& parm);
```

Parameters

p

The stored `p` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 \leq p \leq 1.0$

The first constructor constructs an object whose stored `p` value holds the value *p*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

bernoulli_distribution::param_type

Contains the parameters of the distribution.

```
struct param_type { typedef bernoulli_distribution distribution_type; param_type(double p = 0.5); double p() const;
bool operator==(const param_type& right) const; bool operator!=(const param_type& right) const; };
```

Parameters

p

The stored `p` distribution parameter.

Remarks

Precondition: $0.0 \leq p \leq 1.0$

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

binomial_distribution Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Generates a binomial distribution.

Syntax

```
template<class IntType = int>
class binomial_distribution
{
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructors and reset functions
    explicit binomial_distribution(result_type t = 1, double p = 0.5);
    explicit binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type t() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **int** if none is provided, distributed according to the Binomial Distribution discrete probability function. The following table links to articles about individual members.

binomial_distribution	<code>binomial_distribution::t</code>	<code>binomial_distribution::param</code>
<code>binomial_distribution::operator()</code>	<code>binomial_distribution::p</code>	param_type

The property members `t()` and `p()` return the currently stored distribution parameter values *t* and *p*

respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the binomial distribution discrete probability function, see the Wolfram MathWorld article [Binomial Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int t, const double p, const int& s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::binomial_distribution<> distr(t, p);

    std::cout << std::endl;
    std::cout << "p == " << distr.p() << std::endl;
    std::cout << "t == " << distr.t() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int t_dist = 1;
    double p_dist = 0.5;
    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter an integer value for t distribution (where 0 <= t): ";
    std::cin >> t_dist;
    std::cout << "Enter a double value for p distribution (where 0.0 <= p <= 1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(t_dist, p_dist, samples);
}

```

First run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where  $0 \leq t$ ): 22
Enter a double value for p distribution (where  $0.0 \leq p \leq 1.0$ ): .25
Enter an integer value for a sample count: 100
```

```
p == 0.25
t == 22
Histogram for 100 samples:
 1 :
 2 ::
 3 :::::::::::
 4 :::::::::::
 5 ::::::::::::::::::::::
 6 ::::::::::::::
 7 :::::::::::
 8 ::::::
 9 ::::::
11 :
12 :
```

Second run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where  $0 \leq t$ ): 22
Enter a double value for p distribution (where  $0.0 \leq p \leq 1.0$ ): .5
Enter an integer value for a sample count: 100
```

```
p == 0.5
t == 22
Histogram for 100 samples:
 6 :
 7 ::
 8 :::::::::::
 9 :::::::::::
10 ::::::::::::::
11 ::::::::::::::
12 :::::::::::
13 :::::::::::
14 :::::::::::
15 ::
16 ::
```

Third run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where  $0 \leq t$ ): 22
Enter a double value for p distribution (where  $0.0 \leq p \leq 1.0$ ): .75
Enter an integer value for a sample count: 100
```

```
p == 0.75
t == 22
Histogram for 100 samples:
13 :::
14 :::::::::::
15 ::::::::::::::
16 ::::::::::::::
17 :::::::::::
18 :::::::::::
19 :::::::::::
20 ::::::
21 :
```

Requirements

Header: <random>

Namespace: std

binomial_distribution::binomial_distribution

Constructs the distribution.

```
explicit binomial_distribution(result_type t = 1, double p = 0.5);
explicit binomial_distribution(const param_type& parm);
```

Parameters

t

The `t` distribution parameter.

p

The `p` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0 \leq t$ and $0.0 \leq p \leq 1.0$

The first constructor constructs an object whose stored *p* value holds the value *p* and whose stored *t* value holds the value *t*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

binomial_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef binomial_distribution<result_type> distribution_type;
    param_type(result_type t = 1, double p = 0.5);
    result_type t() const;
    double p() const;
    .....
    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

t

The `t` distribution parameter.

p

The `p` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: $0 \leq t$ and $0.0 \leq p \leq 1.0$

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

cauchy_distribution Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Generates a Cauchy distribution.

Syntax

```
template<class RealType = double>
class cauchy_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit cauchy_distribution(result_type a = 0.0, result_type b = 1.0);
    explicit cauchy_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating-point type, or type **double** if none is provided, distributed according to the Cauchy Distribution. The following table links to articles about individual members.

cauchy_distribution	<code>cauchy_distribution::a</code>	<code>cauchy_distribution::param</code>
<code>cauchy_distribution::operator()</code>	<code>cauchy_distribution::b</code>	param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters `a` and `b`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the cauchy distribution, see the Wolfram MathWorld article [Cauchy Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;

    std::mt19937 gen(1701);

    std::cauchy_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'a' distribution parameter: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'b' distribution parameter (must be greater than zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}

```

First run:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'a' distribution parameter: 0  
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 1  
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308  
max() == 1.79769e+308  
a() == 0.0000000000  
b() == 1.0000000000  
Distribution for 10 samples:  
1: -3.4650392984  
2: -2.6369564174  
3: -0.0786978867  
4: -0.0609632093  
5: 0.0589387400  
6: 0.0589539764  
7: 0.1004592006  
8: 1.0965724260  
9: 1.4389408122  
10: 2.5253154706
```

Second run:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'a' distribution parameter: 0  
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 10  
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308  
max() == 1.79769e+308  
a() == 0.0000000000  
b() == 10.0000000000  
Distribution for 10 samples:  
1: -34.6503929840  
2: -26.3695641736  
3: -0.7869788674  
4: -0.6096320926  
5: 0.5893873999  
6: 0.5895397637  
7: 1.0045920062  
8: 10.9657242597  
9: 14.3894081218  
10: 25.2531547063
```

Third run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 10
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 10
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308
max() == 1.79769e+308
a() == 10.0000000000
b() == 10.0000000000
Distribution for 10 samples:
 1: -24.6503929840
 2: -16.3695641736
 3:  9.2130211326
 4:  9.3903679074
 5: 10.5893873999
 6: 10.5895397637
 7: 11.0045920062
 8: 20.9657242597
 9: 24.3894081218
10: 35.2531547063
```

Requirements

Header: <random>

Namespace: std

cauchy_distribution::cauchy_distribution

Constructs the distribution.

```
explicit cauchy_distribution(result_type a = 0.0, result_type b = 1.0);
explicit cauchy_distribution(const param_type& parm);
```

Parameters

a

The `a` distribution parameter.

b

The `b` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 < b$

The first constructor constructs an object whose stored `a` value holds the value *a* and whose stored `b` value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

cauchy_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef cauchy_distribution<result_type> distribution_type;
    param_type(result_type a = 0.0, result_type b = 1.0);
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

a

The `a` distribution parameter.

b

The `b` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

chi_squared_distribution Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Generates a chi-squared distribution.

Syntax

```
template<class RealType = double>
class chi_squared_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit chi_squared_distribution(RealType n = 1);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating-point type, or type **double** if none is provided, distributed according to the Chi-Squared Distribution. The following table links to articles about individual members.

chi_squared_distribution	<code>chi_squared_distribution::n</code>	<code>chi_squared_distribution::param</code>
<code>chi_squared_distribution::operator()</code>		param_type

The property function `n()` returns the value for the stored distribution parameter `n`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the chi-squared distribution, see the Wolfram MathWorld article [Chi-Squared Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double n, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::chi_squared_distribution<> distr(n);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double n_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the '\n\' distribution parameter (must be greater than
zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(n_dist, samples);
}

```

First run:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): .5  
Enter an integer value for the sample count: 10
```

```
min() == 4.94066e-324  
max() == 1.79769e+308  
n() == 0.5000000000  
Distribution for 10 samples:  
1: 0.0007625595  
2: 0.0016895062  
3: 0.0058683478  
4: 0.0189647765  
5: 0.0556619371  
6: 0.1448191353  
7: 0.1448245325  
8: 0.1903494379  
9: 0.9267525768  
10: 1.5429743723
```

Second run:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): .3333  
Enter an integer value for the sample count: 10
```

```
min() == 4.94066e-324  
max() == 1.79769e+308  
n() == 0.3333000000  
Distribution for 10 samples:  
1: 0.0000148725  
2: 0.0000490528  
3: 0.0003175988  
4: 0.0018454535  
5: 0.0092808795  
6: 0.0389540735  
7: 0.0389562514  
8: 0.0587028468  
9: 0.6183666639  
10: 1.3552086624
```

Third run:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): 1000  
Enter an integer value for the sample count: 10
```

```
min() == 4.94066e-324  
max() == 1.79769e+308  
n() == 1000.0000000000  
Distribution for 10 samples:  
1: 958.5284624473  
2: 958.7882787809  
3: 963.0667684792  
4: 987.9638091514  
5: 1016.2433493745  
6: 1021.9337111110  
7: 1021.9723046240  
8: 1035.7622110505  
9: 1043.8725156645  
10: 1054.7051509381
```

Requirements

Header: <random>

Namespace: std

chi_squared_distribution::chi_squared_distribution

Constructs the distribution.

```
explicit chi_squared_distribution(result_type n = 1.0);
explicit chi_squared_distribution(const param_type& parm);
```

Parameters

n

The `n` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < n`

The first constructor constructs an object whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

chi_squared_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef chi_squared_distribution<result_type> distribution_type;
    param_type(result_type n = 1.0);
    result_type n() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

n

The `n` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < n`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

discard_block_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence by discarding values returned by its base engine.

Syntax

```
template <class Engine, size_t P, size_t R>
class discard_block_engine;
```

Parameters

Engine

The base engine type.

P

Block size. The number of values in each block.

R

Used block. The number of values in each block that are used. The rest are discarded ($P - R$). **Precondition:**

$0 < R \leq P$

Members

<code>discard_block_engine::discard_block_engine</code>	<code>discard_block_engine::base</code>	<code>discard_block_engine::discard</code>
<code>discard_block_engine::operator()</code>	<code>discard_block_engine::base_type</code>	<code>discard_block_engine::seed</code>

For more information about engine members, see [<random>](#).

Remarks

This template class describes an engine adaptor that produces values by discarding some of the values returned by its base engine.

Requirements

Header: `<random>`

Namespace: `std`

See also

[<random>](#)

discrete_distribution Class

11/15/2018 • 5 minutes to read • [Edit Online](#)

Generates a discrete integer distribution that has uniform-width intervals with uniform probability in each interval.

Syntax

```
template<class IntType = int>
class discrete_distribution
{
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructor and reset functions
    discrete_distribution();
    template <class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> weightlist);
    template <class UnaryOperation>
    discrete_distribution(size_t count, double xmin, double xmax, UnaryOperation funcweight);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

Remarks

This sampling distribution has uniform-width intervals with uniform probability in each interval. For information about other sampling distributions, see [piecewise_linear_distribution Class](#) and [piecewise_constant_distribution Class](#).

The following table links to articles about individual members:

discrete_distribution	<code>discrete_distribution::param</code>
---------------------------------------	---

<code>discrete_distribution::operator()</code>	<code>param_type</code>

The property function `vector<double> probabilities()` returns the individual probabilities for each integer generated.

For more information about distribution classes and their members, see [<random>](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    discrete_distribution<> distr({ 1, 2, 3, 4, 5 });

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "probabilities (value: probability):" << endl;
    vector<double> p = distr.probabilities();
    int counter = 0;
    for (const auto& n : p) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
        ++counter;
    }
    cout << endl;

    // generate the distribution as a histogram
    map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << s << " samples:" << endl;
    for (const auto& elem : histogram) {
        cout << setw(5) << elem.first << ' ' << string(elem.second, ':') << endl;
    }
    cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}

```


Use CTRL-Z to bypass data entry and run using default values.

Enter an integer value for the sample count: 100

min() == 0

max() == 4

probabilities (value: probability):

0:	0.0666666667
1:	0.1333333333
2:	0.2000000000
3:	0.2666666667
4:	0.3333333333

Distribution for 100 samples:

0	:::
1	::::::::::::::
2	::::::::::::::
3	::::::::::::::::::::::
4	::::::::::::::::::::::

Requirements

Header: <random>

Namespace: std

discrete_distribution::discrete_distribution

Constructs the distribution.

```
// default constructor
discrete_distribution();

// construct using a range of weights, [firstW, lastW)
template <class InputIterator>
discrete_distribution(InputIterator firstW, InputIterator lastW);

// construct using an initializer list for range of weights
discrete_distribution(initializer_list<double> weightlist);

// construct using unary operation function
template <class UnaryOperation>
discrete_distribution(size_t count, double low, double high, UnaryOperation weightfunc);

// construct from an existing param_type structure
explicit discrete_distribution(const param_type& parm);
```

Parameters

firstW

The first iterator in the list from which to construct the distribution.

lastW

The last iterator in the list from which to construct the distribution (non-inclusive because iterators use an empty element for the end).

weightlist

The [initializer_list](#) from which to construct the distribution.

count

The number of elements in the distribution range. If `count==0`, equivalent to the default constructor (always generates zero).

low

The lowest value in the distribution range.

high

The highest value in the distribution range.

weightfunc

The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to **double**.

parm

The `param_type` structure used to construct the distribution.

Remarks

The default constructor constructs an object whose stored probability value has one element with value 1. This will result in a distribution that always generates a zero.

The iterator range constructor that has parameters *firstW* and *lastW* constructs a distribution object by using weight values taken from the iterators over the interval sequence [*firstW*, *lastW*).

The initializer list constructor that has a *weightlist* parameter constructs a distribution object with weights from the initializer list *weightlist*.

The constructor that has *count*, *low*, *high*, and *weightfunc* parameters constructs a distribution object initialized based on these rules:

- If *count* < 1, **n** = 1, and as such is equivalent to the default constructor, always generating zero.
- If *count* > 0, **n** = *count*. Provided **d** = (*high* - *low*) / **n** is greater than zero, using **d** uniform subranges, each weight is assigned as follows: `weight[k] = weightfunc(x)`, where **x** = *low* + **k** * **d** + **d** / 2, for **k** = 0, ..., **n** - 1.

The constructor that has a `param_type` parameter *parm* constructs a distribution object using *parm* as the stored parameter structure.

discrete_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef discrete_distribution<result_type> distribution_type;
    param_type();

    // construct using a range of weights, [firstW, lastW)
    template <class InputIterator>
    param_type(InputIterator firstW, InputIterator lastW);

    // construct using an initializer list for range of weights
    param_type(initializer_list<double> weightlist);

    // construct using unary operation function
    template <class UnaryOperation>
    param_type(size_t count, double low, double high, UnaryOperation weightfunc);

    std::vector<double> probabilities() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

firstW

The first iterator in the list from which to construct the distribution.

lastW

The last iterator in the list from which to construct the distribution (non-inclusive because iterators use an empty element for the end).

weightlist

The [initializer_list](#) from which to construct the distribution.

count

The number of elements in the distribution range. If *count* is 0, this is equivalent to the default constructor (always generates zero).

low

The lowest value in the distribution range.

high

The highest value in the distribution range.

weightfunc

The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to **double**.

right

The `param_type` object to compare to this.

Remarks

This parameter package can be passed to `operator()` to generate the return value.

See also

[<random>](#)

exponential_distribution Class

11/9/2018 • 3 minutes to read • [Edit Online](#)

Generates an exponential distribution.

Syntax

```
template<class RealType = double>
class exponential_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructors and reset functions
    explicit exponential_distribution(result_type lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **double** if none is provided, distributed according to the Exponential Distribution. The following table links to articles about individual members.

exponential_distribution	<code>exponential_distribution::lambda</code>	<code>exponential_distribution::param</code>
<code>exponential_distribution::operator()</code>		param_type

The property member function `lambda()` returns the value for the stored distribution parameter `lambda`.

The property member function `param()` sets or returns the `param_type` stored distribution parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the exponential distribution, see the Wolfram MathWorld article [Exponential Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double l, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::exponential_distribution<> distr(l);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "lambda() == " << std::fixed << std::setw(11) << distr.lambda() <<
    std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double l_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'lambda' distribution parameter (must be greater than
zero): ";
    std::cin >> l_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(l_dist, samples);
}
```

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'lambda' distribution parameter (must be greater than zero): 1  
Enter an integer value for the sample count: 10
```

```
min() == 0  
max() == 1.79769e+308  
lambda() == 1.0000000000  
Distribution for 10 samples:  
1: 0.0936880533  
2: 0.1225944894  
3: 0.6443593183  
4: 0.6551171649  
5: 0.7313457551  
6: 0.7313557977  
7: 0.7590097389  
8: 1.4466885214  
9: 1.6434088411  
10: 2.1201210996
```

Requirements

Header: <random>

Namespace: std

exponential_distribution::exponential_distribution

Constructs the distribution.

```
explicit exponential_distribution(result_type lambda = 1.0);  
explicit exponential_distribution(const param_type& parm);
```

Parameters

lambda

The `lambda` distribution parameter.

parm

The parameter package used to construct the distribution.

Remarks

Precondition: `0.0 < lambda`

The first constructor constructs an object whose stored `lambda` value holds the value *lambda*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

exponential_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef exponential_distribution<result_type> distribution_type;
    param_type(result_type lambda = 1.0);
    result_type lambda() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

lambda

The `lambda` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < lambda`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

extreme_value_distribution Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Generates an extreme value distribution.

Syntax

```
template<class RealType = double>
class extreme_value_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit extreme_value_distribution(result_type a = 0.0, result_type b = 1.0);
    explicit extreme_value_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating-point type, or type **double** if none is provided, distributed according to the Extreme Value Distribution. The following table links to articles about individual members.

extreme_value_distribution	<code>extreme_value_distribution::a</code>	<code>extreme_value_distribution::param</code>
<code>extreme_value_distribution::operator()</code>	<code>extreme_value_distribution::b</code>	param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters `a` and `b`.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the extreme value distribution, see the Wolfram MathWorld article [Extreme Value Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;

    std::mt19937 gen(1701);

    std::extreme_value_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the \'a\' distribution parameter: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the \'b\' distribution parameter (must be greater than
zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 0
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308
max() == 1.79769e+308
a() == 0.0000000000
b() == 1.0000000000
Distribution for 10 samples:
 1: -0.8813940331
 2: -0.7698972281
 3: 0.2951258007
 4: 0.3110450734
 5: 0.4210546820
 6: 0.4210688771
 7: 0.4598857960
 8: 1.3155194200
 9: 1.5379170046
10: 2.0568757061
```

Requirements

Header: <random>

Namespace: std

extreme_value_distribution::extreme_value_distribution

Constructs the distribution.

```
explicit extreme_value_distribution(result_type a_value = 0.0, result_type b_value = 1.0);
explicit extreme_value_distribution(const param_type& parm);
```

Parameters

a_value

The `a` distribution parameter.

b_value

The `b` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 < b$

The first constructor constructs an object whose stored `a` value holds the value *a_value* and whose stored `b` value holds the value *b_value*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

extreme_value_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef extreme_value_distribution<result_type> distribution_type;
    param_type(result_type a_value = 0.0, result_type b_value = 1.0);
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

a_value

The `a` distribution parameter.

b_value

The `b` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

fisher_f_distribution Class

11/9/2018 • 4 minutes to read • [Edit Online](#)

Generates a Fisher F distribution.

Syntax

```
template<class RealType = double>
class fisher_f_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type; // constructor and reset functions
    explicit fisher_f_distribution(result_type m = 1.0, result_type n = 1.0);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type m() const;
    result_type n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating-point type, or type **double** if none is provided, distributed according to the Fisher's F-Distribution. The following table links to articles about individual members.

fisher_f_distribution	<code>fisher_f_distribution::m</code>	<code>fisher_f_distribution::param</code>
<code>fisher_f_distribution::operator()</code>	<code>fisher_f_distribution::n</code>	param_type

The property functions `m()` and `n()` return the values for the stored distribution parameters `m` and `n` respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the F-distribution, see the Wolfram MathWorld article [F-Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double m, const double n, const int s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1701);

    std::fisher_f_distribution<> distr(m, n);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "m() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.m() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double m_dist = 1;
    double n_dist = 1;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the \'m\' distribution parameter (must be greater than
zero): ";
    std::cin >> m_dist;
    std::cout << "Enter a floating point value for the \'n\' distribution parameter (must be greater than
zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(m_dist, n_dist, samples);
}

```

Output

First run:

```
Enter a floating point value for the 'm' distribution parameter (must be greater than zero): 1
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == 0
max() == 1.79769e+308
m() == 1.0000000000
n() == 1.0000000000
Distribution for 10 samples:
1: 0.0204569549
2: 0.0221376644
3: 0.0297234962
4: 0.1600937252
5: 0.2775342196
6: 0.3950701700
7: 0.8363200295
8: 0.9512500702
9: 2.7844815974
10: 3.4320929653
```

Second run:

```
Enter a floating point value for the 'm' distribution parameter (must be greater than zero): 1
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): .1
Enter an integer value for the sample count: 10
```

```
min() == 0
max() == 1.79769e+308
m() == 1.0000000000
n() == 0.1000000000
Distribution for 10 samples:
1: 0.0977725649
2: 0.5304122767
3: 4.9468518084
4: 25.1012074939
5: 48.8082121613
6: 401.8075539377
7: 8199.5947873699
8: 226492.6855335717
9: 2782062.6639740225
10: 20829747131.7185860000
```

Third run:

```
Enter a floating point value for the 'm' distribution parameter (must be greater than zero): .1
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == 0
max() == 1.79769e+308
m() == 0.1000000000
n() == 1.0000000000
Distribution for 10 samples:
1: 0.0000000000
2: 0.0000000000
3: 0.0000000000
4: 0.0000000000
5: 0.0000000033
6: 0.0000073975
7: 0.0000703800
8: 0.0280427735
9: 0.2660239949
10: 3.4363333954
```

Requirements

Header: <random>

Namespace: std

fisher_f_distribution::fisher_f_distribution

Constructs the distribution.

```
explicit fisher_f_distribution(result_type m = 1.0, result_type n = 1.0);
explicit fisher_f_distribution(const param_type& parm);
```

Parameters

m

The `m` distribution parameter.

n

The `n` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 < m$ and $0.0 < n$

The first constructor constructs an object whose stored `m` value holds the value *m* and whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

fisher_f_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef fisher_f_distribution<result_type> distribution_type;
    param_type(result_type m = 1.0, result_type n = 1.0);
    result_type m() const;
    result_type n() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

m

The `m` distribution parameter.

n

The `n` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: $0.0 < m$ and $0.0 < n$

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

gamma_distribution Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Generates a gamma distribution.

Syntax

```
template<class RealType = double>
class gamma_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructors and reset functions
    explicit gamma_distribution(result_type alpha = 1.0, result_type beta = 1.0);
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type alpha() const;
    result_type beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating-point type, or type **double** if none is provided, distributed according to the Gamma Distribution. The following table links to articles about individual members.

gamma_distribution	<code>gamma_distribution::alpha</code>	<code>gamma_distribution::param</code>
<code>gamma_distribution::operator()</code>	<code>gamma_distribution::beta</code>	param_type

The property functions `alpha()` and `beta()` return their respective values for stored distribution parameters *alpha* and *beta*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the gamma distribution, see the Wolfram MathWorld article [Gamma Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;

    std::mt19937 gen(1701);

    std::gamma_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "alpha() == " << std::fixed << std::setw(11) << distr.alpha() <<
std::endl;
    std::cout << "beta() == " << std::fixed << std::setw(11) << distr.beta() <<
std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'alpha' distribution parameter (must be greater than
zero): ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'beta' distribution parameter (must be greater than
zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}

```

Use CTRL-Z to bypass data entry and run using default values.

Enter a floating point value for the 'alpha' distribution parameter (must be greater than zero): 1

Enter a floating point value for the 'beta' distribution parameter (must be greater than zero): 1

Enter an integer value for the sample count: 10

```
min() == 4.94066e-324
```

```
max() == 1.79769e+308
```

```
alpha() == 1.0000000000
```

```
beta() == 1.0000000000
```

```
Distribution for 10 samples:
```

```
1: 0.0936880533
```

```
2: 0.1225944894
```

```
3: 0.6443593183
```

```
4: 0.6551171649
```

```
5: 0.7313457551
```

```
6: 0.7313557977
```

```
7: 0.7590097389
```

```
8: 1.4466885214
```

```
9: 1.6434088411
```

```
10: 2.1201210996
```

Requirements

Header: <random>

Namespace: std

gamma_distribution::gamma_distribution

Constructs the distribution.

```
explicit gamma_distribution(result_type alpha = 1.0, result_type beta = 1.0);
explicit gamma_distribution(const param_type& parm);
```

Parameters

alpha

The `alpha` distribution parameter.

beta

The `beta` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < alpha` and `0.0 < beta`

The first constructor constructs an object whose stored `alpha` value holds the value *alpha* and whose stored `beta` value holds the value *beta*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

gamma_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef gamma_distribution<result_type> distribution_type;
    param_type(result_type alpha = 1.0, result_type beta 1.0);
    result_type alpha() const;
    result_type beta() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

alpha

The `alpha` distribution parameter.

beta

The `beta` distribution parameter.

right

The `param_type` instance to compare this to.

Remarks

Precondition: `0.0 < alpha` and `0.0 < beta`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

geometric_distribution Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Generates a geometric distribution.

Syntax

```
template<class IntType = int>
class geometric_distribution {
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructors and reset functions
    explicit geometric_distribution(double p = 0.5);
    explicit geometric_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

URNG

The uniform random number generator engine. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type with a geometric distribution. The following table links to articles about individual members.

geometric_distribution	<code>geometric_distribution::p</code>	<code>geometric_distribution::param</code>
<code>geometric_distribution::operator()</code>		param_type

The property function `p()` returns the value for stored distribution parameter `p`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the chi-squared distribution, see the Wolfram MathWorld article [Geometric Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::geometric_distribution<> distr(p);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "p() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 0.5;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the \'p\' distribution parameter: ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```


First test:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'p' distribution parameter: .5  
Enter an integer value for the sample count: 100
```

```
min() == 0  
max() == 2147483647  
p() == 0.5000000000  
Distribution for 100 samples:  
  0 :::::::::::::::::::::::::::::::::::::::  
  1 :::::::::::::::  
  2 :::::::::::  
  3 :::::::  
  4 ::  
  5 :
```

Second test:

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'p' distribution parameter: .1  
Enter an integer value for the sample count: 100
```

```
min() == 0  
max() == 2147483647  
p() == 0.1000000000  
Distribution for 100 samples:  
  0 :::::::::::  
  1 :::::::::::  
  2 :::::::::::  
  3 :::::::::::  
  4 :::::::  
  5 :::::::::::  
  6 :::  
  7 :::::::  
  8 :::::::::::  
  9 :::::::  
 10 :::  
 11 :::  
 12 ::  
 13 :  
 14 :::  
 15 ::  
 16 :::  
 17 :::  
 20 ::::::  
 21 :  
 29 :  
 32 :  
 35 :
```

Requirements

Header: <random>

Namespace: std

geometric_distribution::geometric_distribution

Constructs the distribution.

```
explicit geometric_distribution(double p = 0.5);
explicit geometric_distribution(const param_type& parm);
```

Parameters

p

The `p` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < p && p < 1.0`

The first constructor constructs an object whose stored `p` value holds the value *p*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

geometric_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef geometric_distribution<result_type> distribution_type;
    param_type(double p = 0.5);
    double p() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

p

The `p` distribution parameter.

right

The `param_type` instance to compare this to.

Remarks

Precondition: `0.0 < p && p < 1.0`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

independent_bits_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence of numbers with a specified number of bits by repacking bits from the values returned by its base engine.

Syntax

```
template <class Engine, size_t W, class UIntType>
class independent_bits_engine;
```

Parameters

Engine

The base engine type.

W

Word size. Size, in bits, of each number generated. **Precondition:** $0 < W \leq \text{numeric_limits}<\text{UIntType}>::\text{digits}$

UIntType

The unsigned integer result type. For possible types, see [<random>](#).

Members

<code>independent_bits_engine::independent_b</code>	<code>independent_bits_engine::base</code>	<code>independent_bits_engine::discard</code>
<code>independent_bits_engine::operator()</code>	<code>independent_bits_engine::base_type</code>	<code>independent_bits_engine::seed</code>

For more information about engine members, see [<random>](#).

Remarks

This template class describes an *engine adaptor* that produces values by repacking bits from the values returned by its base engine, resulting in *W*-bit values.

Requirements

Header: [<random>](#)

Namespace: std

See also

[<random>](#)

linear_congruential_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence by the linear congruential algorithm.

Syntax

```
class linear_congruential_engine{
public: // types
typedef UIntType result_type;
// engine characteristics
static constexpr result_type multiplier = a;
static constexpr result_type increment = c;
static constexpr result_type modulus = m;
static constexpr result_type min() { return c == 0u ? 0u : 0u; }
static constexpr result_type max() { return m - 1u; }
static constexpr result_type default_seed = 1u;
// constructors and seeding functions
explicit linear_congruential_engine(result_type s = default_seed);
template <class Sseq>
explicit linear_congruential_engine(Sseq& q);
void seed(result_type s = default_seed);
template <class Sseq>
void seed(Sseq& q);
// generating functions
result_type operator()();
void discard(unsigned long long z);
};
```

Parameters

UIntType

The unsigned integer result type. For possible types, see [<random>](#).

A

Multiplier. Precondition: See Remarks section.

C

Increment. Precondition: See Remarks section.

M

Modulus. Precondition: See remarks.

Members

<code>linear_congruential_engine::linear_con</code>	<code>linear_congruential_engine::min</code>	<code>linear_congruential_engine::discard</code>
<code>linear_congruential_engine::operator()</code>	<code>linear_congruential_engine::max</code>	<code>linear_congruential_engine::seed</code>

`default_seed` is a member constant, defined as `1u`, used as the default parameter value for `linear_congruential_engine::seed` and the single value constructor.

For more information about engine members, see [<random>](#).

Remarks

The `linear_congruential_engine` template class is the simplest generator engine, but not the fastest or highest quality. An improvement over this engine is the [subtract_with_carry_engine](#). Neither of these engines is as fast or with as high quality results as the [mersenne_twister_engine](#).

This engine produces values of a user-specified unsigned integral type using the recurrence relation (*period*)

```
x(i) = (A * x(i-1) + C) mod M.
```

If M is zero, the value used for this modulus operation is `numeric_limits<result_type>::max() + 1`. The engine's state is the last value returned, or the seed value if no call has been made to `operator()`.

If M is not zero, the values of the template arguments A and C must be less than M .

Although you can construct a generator from this engine directly, you can also use one of these predefined typedefs.

`minstd_rand0`: 1988 minimal standard engine (Lewis, Goodman, and Miller, 1969).

```
typedef linear_congruential_engine<unsigned int, 16807, 0, 2147483647> minstd_rand0;
```

`minstd_rand`: Updated minimal standard engine `minstd_rand0` (Park, Miller, and Stockmeyer, 1993).

```
typedef linear_congruential_engine<unsigned int, 48271, 0, 2147483647> minstd_rand;
```

For detailed information about the linear congruential engine algorithm, see the Wikipedia article [Linear congruential generator](#).

Requirements

Header: `<random>`

Namespace: `std`

See also

[<random>](#)

lognormal_distribution Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Generates a log normal distribution.

Syntax

```
template <class RealType = double>
class lognormal_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;
    // constructor and reset functions
    explicit lognormal_distribution(result_type m = 0.0, result_type s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();
    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);
    // property functions
    result_type m() const;
    result_type s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **double** if none is provided, distributed according to the Log Normal Distribution. The following table links to articles about individual members.

lognormal_distribution	<code>lognormal_distribution::m</code>	<code>lognormal_distribution::param</code>
<code>lognormal_distribution::operator()</code>	<code>lognormal_distribution::s</code>	param_type

The property functions `m()` and `s()` return the values for the stored distribution parameters *m* and *s*, respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the LogNormal distribution, see the Wolfram MathWorld article [LogNormal Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const double m, const double s, const int samples) {

    // uncomment to use a non-deterministic seed
    // random_device gen;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    lognormal_distribution<> distr(m, s);

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "m() == " << fixed << setw(11) << setprecision(10) << distr.m() << endl;
    cout << "s() == " << fixed << setw(11) << setprecision(10) << distr.s() << endl;

    // generate the distribution as a histogram
    map<double, int> histogram;
    for (int i = 0; i < samples; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << samples << " samples:" << endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        cout << fixed << setw(11) << ++counter << ": "
            << setw(14) << setprecision(10) << elem.first << endl;
    }
    cout << endl;
}

int main()
{
    double m_dist = 1;
    double s_dist = 1;
    int samples = 10;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter a floating point value for the 'm' distribution parameter: ";
    cin >> m_dist;
    cout << "Enter a floating point value for the 's' distribution parameter (must be greater than zero): ";
    cin >> s_dist;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(m_dist, s_dist, samples);
}

```



```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'm' distribution parameter: 0
Enter a floating point value for the 's' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308
max() == 1.79769e+308
m() == 0.0000000000
s() == 1.0000000000
Distribution for 10 samples:
 1: 0.3862809339
 2: 0.4128865601
 3: 0.4490576787
 4: 0.4862035428
 5: 0.5930607126
 6: 0.8190778771
 7: 0.8902379317
 8: 2.8332911667
 9: 5.1359445565
10: 5.4406507912
```

Requirements

Header: <random>

Namespace: std

lognormal_distribution::lognormal_distribution

Constructs the distribution.

```
explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
explicit lognormal_distribution(const param_type& parm);
```

Parameters

m

The `m` distribution parameter.

s

The `s` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 < s$

The first constructor constructs an object whose stored `m` value holds the value *m* and whose stored `s` value holds the value *s*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

lognormal_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef lognormal_distribution<result_type> distribution_type;
    param_type(result_type m = 0.0, result_type s = 1.0);
    result_type m() const;
    result_type s() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

m

The `m` distribution parameter.

s

The `s` distribution parameter.

right

The `param_type` structure used to compare.

Remarks

Precondition: `0.0 < s`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

mersenne_twister_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a high quality random sequence of integers based on the Mersenne twister algorithm.

Syntax

```
template <class UIntType,
    size_t W, size_t N, size_t M, size_t R,
    UIntType A, size_t U, UIntType D, size_t S,
    UIntType B, size_t T, UIntType C, size_t L, UIntType F>
class mersenne_twister_engine;
```

Parameters

UIntType

The unsigned integer result type. For possible types, see [<random>](#).

W

Word size. Size of each word, in bits, of the state sequence. **Precondition:**

$2u < W \leq \text{numeric_limits}<\text{UIntType}>::\text{digits}$

N

State size. The number of elements (values) in the state sequence.

M

Shift size. The number of elements to skip during each twist. **Precondition:** $0 < M \leq N$

R

Mask bits. Precondition: $R \leq W$

A

XOR mask. Precondition: $A \leq (1u << W) - 1u$

U, S, T, L

Tempering shift parameters. Used as shift values during scrambling (tempering). Precondition: $U, S, T, L \leq W$

D, B, C

Tempering bit mask parameters. Used as bit mask values during scrambling (tempering). Precondition:

$D, B, C \leq (1u << W) - 1u$

F

Initialization multiplier. Used to help with initialization of the sequence. Precondition: $F \leq (1u << W) - 1u$

Members

<code>mersenne_twister_engine::mersenne_twis</code>	<code>mersenne_twister_engine::min</code>	<code>mersenne_twister_engine::discard</code>
<code>mersenne_twister_engine::operator()</code>	<code>mersenne_twister_engine::max</code>	<code>mersenne_twister_engine::seed</code>

`default_seed` is a member constant, defined as `5489u`, used as the default parameter value for

`mersenne_twister_engine::seed` and the single value constructor.

For more information about engine members, see [<random>](#).

Remarks

This template class describes a random number engine, returning values on the closed interval $[0, 2^W - 1]$. It holds a large integral value with $W * (N - 1) + R$ bits. It extracts W bits at a time from this large value, and when it has used all the bits it twists the large value by shifting and mixing the bits so that it has a new set of bits to extract from. The engine's state is the last N W -bit values used if `operator()` has been called at least N times, otherwise the M W -bit values that have been used and the last $N - M$ values of the seed.

The generator twists the large value that it holds by using a twisted generalized feedback shift register defined by shift values N and M , a twist value R , and a conditional XOR-mask A . Additionally, the bits of the raw shift register are scrambled (tempered) according to a bit-scrambling matrix defined by values U, D, S, B, T, C , and L .

The template argument `UIntType` must be large enough to hold values up to $2^W - 1$. The values of the other template arguments must satisfy the following requirements:

```
2u < W, 0 < M, M ≤ N, R ≤ W, U ≤ W, S ≤ W, T ≤ W, L ≤ W, W ≤ numeric_limits<UIntType>::digits, A ≤ (1u<<W) - 1u, B ≤ (1u<<W) - 1u, C ≤ (1u<<W) - 1u, D ≤ (1u<<W) - 1u, and F ≤ (1u<<W) - 1u
```

.

Although you can construct a generator from this engine directly, it is recommended you use one of these predefined typedefs:

`mt19937`: 32-bit Mersenne twister engine (Matsumoto and Nishimura, 1998).

```
typedef mersenne_twister_engine<unsigned int, 32, 624, 397,
    31, 0x9908b0df,
    11, 0xffffffff,
    7, 0x9d2c5680,
    15, 0xefc60000,
    18, 1812433253> mt19937;
```

`mt19937_64`: 64-bit Mersenne twister engine (Matsumoto and Nishimura, 2000).

```
typedef mersenne_twister_engine<unsigned long long, 64, 312, 156,
    31, 0xb5026f5aa96619e9ULL,
    29, 0x5555555555555555ULL,
    17, 0x71d67fffed60000ULL,
    37, 0xfff7eee000000000ULL,
    43, 6364136223846793005ULL> mt19937_64;
```

For detailed information about the Mersenne twister algorithm, see the Wikipedia article [Mersenne twister](#).

Example

For a code example, see [<random>](#).

Requirements

Header: `<random>`

Namespace: `std`

See also

<random>

negative_binomial_distribution Class

11/9/2018 • 4 minutes to read • [Edit Online](#)

Generates a negative binomial distribution.

Syntax

```
template<class IntType = int>
class negative_binomial_distribution
{
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructor and reset functions
    explicit negative_binomial_distribution(result_type k = 1, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template `<`class URNG>
    result_type operator()(URNG& gen);
    template `<`class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **int** if none is provided, distributed according to the Negative Binomial Distribution discrete probability function. The following table links to articles about individual members.

negative_binomial_distribution	<code>negative_binomial_distribution::k</code>	<code>negative_binomial_distribution::param</code>
<code>negative_binomial_distribution::operator</code>	<code>negative_binomial_distribution::p</code>	param_type

The property members `k()` and `p()` return the currently stored distribution parameter values *k* and *p* respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the negative binomial distribution discrete probability function, see the Wolfram MathWorld article [Negative Binomial Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int k, const double p, const int& s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::negative_binomial_distribution<> distr(k, p);

    std::cout << std::endl;
    std::cout << "k == " << distr.k() << std::endl;
    std::cout << "p == " << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int k_dist = 1;
    double p_dist = 0.5;
    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter an integer value for k distribution (where 0 < k): ";
    std::cin >> k_dist;
    std::cout << "Enter a double value for p distribution (where 0.0 < p <= 1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(k_dist, p_dist, samples);
}

```

First run:


```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for k distribution (where 0 <= k): 1
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .5
Enter an integer value for a sample count: 100
```

```
k == 1
p == 0.5
Histogram for 100 samples:
 0 ::::::::::::::::::::::::::::::::::::::
 1 ::::::::::::::::::::::::::::::::::::::
 2 ::::::::::::::
 3 :::::::
 4 ::::
 5 ::
```

Second run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for k distribution (where 0 <= k): 100
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .667
Enter an integer value for a sample count: 100
```

```
k == 100
p == 0.667
Histogram for 100 samples:
31 ::
32 :
33 ::
34 :
35 ::
37 ::
38 :
39 :
40 ::
41 :::
42 :::
43 ::::
44 ::::
45 ::::
46 :::::
47 :::::
48 ::
49 ::
50 :::::
51 :::::
52 ::
53 ::
54 ::::
56 ::::
58 :
59 ::::
60 ::
61 :
62 ::
64 :
69 :::
```

Requirements

Header: <random>

Namespace: std

negative_binomial_distribution::negative_binomial_distribution

Constructs the distribution.

```
explicit negative_binomial_distribution(result_type k = 1, double p = 0.5);
explicit negative_binomial_distribution(const param_type& parm);
```

Parameters

k

The `k` distribution parameter.

p

The `p` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: $0.0 < k$ and $0.0 < p \leq 1.0$

The first constructor constructs an object whose stored `p` value holds the value p and whose stored `k` value holds the value k .

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

negative_binomial_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type { typedef negative_binomial_distribution< result_type> distribution_type;
param_type(result_type k = 1, double p = 0.5); result_type k() const; double p() const;

bool operator==(const param_type& right) const; bool operator!=(const param_type& right) const; };
```

Parameters

k

The `k` distribution parameter.

p

The `p` distribution parameter.

right

The `param_type` structure used to compare.

Remarks

Precondition: $0.0 < k$ and $0.0 < p \leq 1.0$

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

normal_distribution Class

3/11/2019 • 4 minutes to read • [Edit Online](#)

Generates a normal distribution.

Syntax

```
template<class RealType = double>
class normal_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructors and reset functions
    explicit normal_distribution(result_type mean = 0.0, result_type stddev = 1.0);
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type mean() const;
    result_type stddev() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **double** if none is provided, distributed according to the Normal Distribution. The following table links to articles about individual members.

normal_distribution	<code>normal_distribution::mean</code>	<code>normal_distribution::param</code>
<code>normal_distribution::operator()</code>	<code>normal_distribution::stddev</code>	param_type

The property functions `mean()` and `stddev()` return the values for the stored distribution parameters *mean* and *stddev* respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the Normal distribution, see the Wolfram MathWorld article [Normal Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const double m, const double s, const int samples) {

    // uncomment to use a non-deterministic seed
    // random_device gen;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    normal_distribution<> distr(m, s);

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "m() == " << fixed << setw(11) << setprecision(10) << distr.mean() << endl;
    cout << "s() == " << fixed << setw(11) << setprecision(10) << distr.stddev() << endl;

    // generate the distribution as a histogram
    map<double, int> histogram;
    for (int i = 0; i < samples; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << samples << " samples:" << endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        cout << fixed << setw(11) << ++counter << ": "
            << setw(14) << setprecision(10) << elem.first << endl;
    }
    cout << endl;
}

int main()
{
    double m_dist = 1;
    double s_dist = 1;
    int samples = 10;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter a floating point value for the 'mean' distribution parameter: ";
    cin >> m_dist;
    cout << "Enter a floating point value for the 'stddev' distribution parameter (must be greater than zero): ";
    cin >> s_dist;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(m_dist, s_dist, samples);
}

```

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter: 0
Enter a floating point value for the 'stddev' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308
max() == 1.79769e+308
m() == 0.0000000000
s() == 1.0000000000
Distribution for 10 samples:
 1: -0.8845823965
 2: -0.1995761116
 3: -0.1162665130
 4: -0.0685154932
 5: 0.0403741461
 6: 0.1591327792
 7: 1.0414389924
 8: 1.5876269426
 9: 1.6362637713
10: 2.7821317338
```

Requirements

Header: <random>

Namespace: std

normal_distribution::normal_distribution

Constructs the distribution.

```
explicit normal_distribution(result_type mean = 0.0, result_type stddev = 1.0);
explicit normal_distribution(const param_type& parm);
```

Parameters

mean

The `mean` distribution parameter.

stddev

The `stddev` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < stddev`

The first constructor constructs an object whose stored `mean` value holds the value *mean* and whose stored `stddev` value holds the value *stddev*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

normal_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef normal_distribution<result_type> distribution_type;
    param_type(result_type mean = 0.0, result_type stddev = 1.0);
    result_type mean() const;
    result_type stddev() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

mean

The `mean` distribution parameter.

stddev

The `stddev` distribution parameter.

right

The `param_type` structure used to compare.

Remarks

Precondition: `0.0 < stddev`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

piecewise_constant_distribution Class

11/15/2018 • 5 minutes to read • [Edit Online](#)

Generates a piecewise constant distribution that has varying-width intervals with uniform probability in each interval.

Syntax

```
template<class RealType = double>
class piecewise_constant_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    piecewise_constant_distribution();
    template <class InputIteratorI, class InputIteratorW>
    piecewise_constant_distribution(
        InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);
    template <class UnaryOperation>
    piecewise_constant_distribution(
        initializer_list<result_type> intervals, UnaryOperation weightfunc);
    template <class UnaryOperation>
    piecewise_constant_distribution(
        size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

This sampling distribution has varying-width intervals with uniform probability in each interval. For information about other sampling distributions, see [piecewise_linear_distribution Class](#) and [discrete_distribution](#).

The following table links to articles about individual members:

<code>piecewise_constant_distribution</code>	<code>piecewise_constant_distribution::inter</code>	<code>piecewise_constant_distribution::param</code>
<code>piecewise_constant_distribution::opera</code>	<code>piecewise_constant_distribution::densi</code>	<code>param_type</code>

The property function `intervals()` returns a `vector<result_type>` with the set of stored intervals of the distribution.

The property function `densities()` returns a `vector<result_type>` with the stored densities for each interval set, which are calculated according to the weights provided in the constructor parameters.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    // Three intervals, non-uniform: 0 to 1, 1 to 6, and 6 to 15
    vector<double> intervals{ 0, 1, 6, 15 };
    // weights determine the densities used by the distribution
    vector<double> weights{ 1, 5, 10 };

    piecewise_constant_distribution<double> distr(intervals.begin(), intervals.end(), weights.begin());

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "intervals (index: interval):" << endl;
    vector<double> i = distr.intervals();
    int counter = 0;
    for (const auto& n : i) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
        ++counter;
    }
    cout << endl;
    cout << "densities (index: density):" << endl;
    vector<double> d = distr.densities();
```

```

counter = 0;
for (const auto& n : d) {
    cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
    ++counter;
}
cout << endl;

// generate the distribution as a histogram
map<int, int> histogram;
for (int i = 0; i < s; ++i) {
    ++histogram[distr(gen)];
}

// print results
cout << "Distribution for " << s << " samples:" << endl;
for (const auto& elem : histogram) {
    cout << setw(5) << elem.first << '-' << elem.first+1 << ' ' << string(elem.second, ':') << endl;
}
cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}

```

Use CTRL-Z to bypass data entry and run using default values.

Enter an integer value for the sample count: 100

min() == 0

max() == 15

intervals (index: interval):

0: 0.0000000000

1: 1.0000000000

2: 6.0000000000

3: 15.0000000000

densities (index: density):

0: 0.0625000000

1: 0.0625000000

2: 0.0694444444

Distribution for 100 samples:

0-1 :::::

1-2 :::::

2-3 :::::

3-4 :::::

4-5 :::::

5-6 :::::

6-7 :::

7-8 :::::

8-9 :::::

9-10 :::::

10-11 :::::

11-12 :::::

12-13 :::::

13-14 :::

14-15 :::::

Requirements

Header: <random>

Namespace: std

piecewise_constant_distribution::piecewise_constant_distribution

Constructs the distribution.

```
// default constructor
piecewise_constant_distribution();

// constructs using a range of intervals, [firstI, lastI), with
// matching weights starting at firstW
template <class InputIteratorI, class InputIteratorW>
piecewise_constant_distribution(InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);

// constructs using an initializer list for range of intervals,
// with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType>
intervals, UnaryOperation weightfunc);

// constructs using an initializer list for range of count intervals,
// distributed uniformly over [xmin,xmax] with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_constant_distribution(size_t count, RealType xmin, RealType xmax, UnaryOperation weightfunc);

// constructs from an existing param_type structure
explicit piecewise_constant_distribution(const param_type& parm);
```

Parameters

firstI

An input iterator of the first element in the distribution range.

lastI

An input iterator of the last element in the distribution range.

firstW

An input iterator of the first element in the weights range.

intervals

An [initializer_list](#) with the intervals of the distribution.

count

The number of elements in the distribution range.

xmin

The lowest value in the distribution range.

xmax

The highest value in the distribution range. Must be greater than *xmin*.

weightfunc

The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to **double**.

parm

The parameter structure used to construct the distribution.

Remarks

The default constructor sets the stored parameters such that there is one interval, 0 to 1, with a probability density of 1.

The iterator range constructor

```
template <class InputIteratorI, class InputIteratorW>
piecewise_constant_distribution(InputIteratorI firstI, InputIteratorI lastI,
                               InputIteratorW firstW);
```

constructs a distribution object with intervals from iterators over the sequence [`firstI`, `lastI`) and a matching weight sequence starting at `firstW`.

The initializer list constructor

```
template <class UnaryOperation>
piecewise_constant_distribution(initializer_list<result_type>
                               intervals,
                               UnaryOperation weightfunc);
```

constructs a distribution object with intervals from the initializer list *intervals* and weights generated from the function *weightfunc*.

The constructor defined as

```
template <class UnaryOperation>
piecewise_constant_distribution(size_t count, result_type xmin, result_type xmax,
                               UnaryOperation weightfunc);
```

constructs a distribution object with *count* intervals distributed uniformly over [`xmin`, `xmax`], assigning each interval weights according to the function *weightfunc*, and *weightfunc* must accept one parameter and have a return value, both of which are convertible to `double`. **Precondition:** `xmin < xmax`

The constructor defined as

```
explicit piecewise_constant_distribution(const param_type& parm);
```

constructs a distribution object using *parm* as the stored parameter structure.

piecewise_constant_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef piecewise_constant_distribution<result_type> distribution_type;
    param_type();
    template <class IterI, class IterW>
    param_type(IterI firstI, IterI lastI, IterW firstW);
    template <class UnaryOperation>
    param_type(size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
    std::vector<result_type> densities() const;
    std::vector<result_type> intervals() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

See the constructor parameters for the [piecewise_constant_distribution](#).

Remarks

Precondition: `xmin < xmax`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

[piecewise_linear_distribution](#)

piecewise_linear_distribution Class

11/15/2018 • 5 minutes to read • [Edit Online](#)

Generates a piecewise linear distribution that has varying-width intervals with linearly varying probability in each interval.

Syntax

```
template<class RealType = double>
class piecewise_linear_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    piecewise_linear_distribution();
    template <class InputIteratorI, class InputIteratorW>
    piecewise_linear_distribution(
        InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);
    template <class UnaryOperation>
    piecewise_linear_distribution(
        initializer_list<result_type> intervals, UnaryOperation weightfunc);
    template <class UnaryOperation>
    piecewise_linear_distribution(
        size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
    explicit piecewise_linear_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

This sampling distribution has varying-width intervals with linearly varying probability in each interval. For information about other sampling distributions, see [piecewise_linear_distribution](#) and [discrete_distribution](#).

The following table links to articles about individual members:

piecewise_linear_distribution	piecewise_linear_distribution::interval	piecewise_linear_distribution::param
piecewise_linear_distribution::operator	piecewise_linear_distribution::densities	param_type

The property function `intervals()` returns a `vector<result_type>` with the set of stored intervals of the distribution.

The property function `densities()` returns a `vector<result_type>` with the stored densities for each interval set, which are calculated according to the weights provided in the constructor parameters.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    // Three intervals, non-uniform: 0 to 1, 1 to 6, and 6 to 15
    vector<double> intervals{ 0, 1, 6, 15 };
    // weights determine the densities used by the distribution
    vector<double> weights{ 1, 5, 5, 10 };

    piecewise_linear_distribution<double> distr(intervals.begin(), intervals.end(), weights.begin());

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "intervals (index: interval):" << endl;
    vector<double> i = distr.intervals();
    int counter = 0;
    for (const auto& n : i) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
        ++counter;
    }
    cout << endl;
    cout << "densities (index: density):" << endl;
    vector<double> d = distr.densities();
```

```

counter = 0;
for (const auto& n : d) {
    cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
    ++counter;
}
cout << endl;

// generate the distribution as a histogram
map<int, int> histogram;
for (int i = 0; i < s; ++i) {
    ++histogram[distr(gen)];
}

// print results
cout << "Distribution for " << s << " samples:" << endl;
for (const auto& elem : histogram) {
    cout << setw(5) << elem.first << '-' << elem.first + 1 << ' ' << string(elem.second, ':') << endl;
}
cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}

```

Use CTRL-Z to bypass data entry and run using default values.

Enter an integer value for the sample count: 100

min() == 0

max() == 15

intervals (index: interval):

0: 0.0000000000

1: 1.0000000000

2: 6.0000000000

3: 15.0000000000

densities (index: density):

0: 0.0645161290

1: 0.3225806452

2: 0.3225806452

3: 0.6451612903

Distribution for 100 samples:

0-1 ::::::::::::::::::::

1-2 :::::

2-3 ::

3-4 :::::

4-5 :::::

5-6 :::::

6-7 :::::

7-8 :::::::::::

8-9 :::::::::::

9-10 :::::

10-11 :::

11-12 ::

12-13 ::

13-14 :::::

14-15 :::::

Requirements

Header: <random>

Namespace: std

piecewise_linear_distribution::piecewise_linear_distribution

Constructs the distribution.

```
// default constructor
piecewise_linear_distribution();

// constructs using a range of intervals, [firstI, lastI), with
// matching weights starting at firstW
template <class InputIteratorI, class InputIteratorW>
piecewise_linear_distribution(InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);

// constructs using an initializer list for range of intervals,
// with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_linear_distribution(initializer_list<RealType>
intervals, UnaryOperation weightfunc);

// constructs using an initializer list for range of count intervals,
// distributed uniformly over [xmin,xmax] with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_linear_distribution(size_t count, RealType xmin, RealType xmax, UnaryOperation weightfunc);

// constructs from an existing param_type structure
explicit piecewise_linear_distribution(const param_type& parm);
```

Parameters

firstI

An input iterator of the first element in the distribution range.

lastI

An input iterator of the last element in the distribution range.

firstW

An input iterator of the first element in the weights range.

intervals

An [initializer_list](#) with the intervals of the distribution.

count

The number of elements in the distribution range.

xmin

The lowest value in the distribution range.

xmax

The highest value in the distribution range. Must be greater than *xmin*.

weightfunc

The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to **double**.

parm

The parameter structure used to construct the distribution.

Remarks

The default constructor sets the stored parameters such that there is one interval, 0 to 1, with a probability density of 1.

The iterator range constructor

```
template <class InputIteratorI, class InputIteratorW>
piecewise_linear_distribution(
    InputIteratorI firstI,
    InputIteratorI lastI,
    InputIteratorW firstW);
```

constructs a distribution object with intervals from iterators over the sequence [`firstI` , `lastI`) and a matching weight sequence starting at *firstW*.

The initializer list constructor

```
template <class UnaryOperation>
piecewise_linear_distribution(
    initializer_list<result_type> intervals,
    UnaryOperation weightfunc);
```

constructs a distribution object with intervals from the initializer list *intervals* and weights generated from the function *weightfunc*.

The constructor defined as

```
template <class UnaryOperation>
piecewise_linear_distribution(
    size_t count,
    result_type xmin,
    result_type xmax,
    UnaryOperation weightfunc);
```

constructs a distribution object with *count* intervals distributed uniformly over [`xmin`, `xmax`], assigning each interval weights according to the function *weightfunc*, and *weightfunc* must accept one parameter and have a return value, both of which are convertible to `double`. **Precondition:** `xmin < xmax`.

The constructor defined as

```
explicit piecewise_linear_distribution(const param_type& parm);
```

constructs a distribution object using *parm* as the stored parameter structure.

piecewise_linear_distribution::param_type

Stores all the parameters of the distribution.

```

struct param_type {
    typedef piecewise_linear_distribution<result_type> distribution_type;
    param_type();
    template <class IterI, class IterW>
    param_type(
        IterI firstI, IterI lastI, IterW firstW);
    template <class UnaryOperation>
    param_type(
        size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
    std::vector<result_type> densities() const;
    std::vector<result_type> intervals() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};

```

Parameters

See constructor parameters for [piecewise_linear_distribution](#).

Remarks

Precondition: `xmin < xmax`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

poisson_distribution Class

11/9/2018 • 3 minutes to read • [Edit Online](#)

Generates a Poisson distribution.

Syntax

```
template<class IntType = int>
class poisson_distribution
{
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    explicit poisson_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    double mean() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type with a Poisson distribution. The following table links to articles about individual members.

poisson_distribution	<code>poisson_distribution::mean</code>	<code>poisson_distribution::param</code>
<code>poisson_distribution::operator()</code>		param_type

The property function `mean()` returns the value for stored distribution parameter *mean*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the Poisson distribution, see the Wolfram MathWorld article [Poisson Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::poisson_distribution<> distr(p);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "p() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.mean() <<
    std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 1.0;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'mean' distribution parameter (must be greater than
zero): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```

First test:

```

Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 100
min() == 0
max() == 2147483647
p() == 1.0000000000
Distribution for 100 samples:
 0 ::::::::::::::::::::::::::::
 1 ::::::::::::::::::::::::::::::
 2 ::::::::::::::::::::::::::::
 3 ::::::::::
 5 :

```

Second test:

```

Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter (must be greater than zero): 10
Enter an integer value for the sample count: 100
min() == 0
max() == 2147483647
p() == 10.0000000000
Distribution for 100 samples:
 3 :
 4 ::
 5 ::
 6 ::::::::::
 7 ::::
 8 ::::::::::
 9 ::::::::::::::
10 ::::::::::::::
11 ::::::::::::::
12 ::::::::::::::
13 ::::::::::
14 ::::::
15 :
16 ::
17 :

```

Requirements

Header: <random>

Namespace: std

poisson_distribution::poisson_distribution

Constructs the distribution.

```

explicit poisson_distribution(RealType mean = 1.0);
explicit binomial_distribution(const param_type& parm);

```

Parameters

mean

The `mean` distribution parameter.

parm

The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < mean`

The first constructor constructs an object whose stored `mean` value holds the value *mean*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

poisson_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef poisson_distribution<IntType> distribution_type;
    param_type(double mean = 1.0);
    double mean() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

See constructor parameters for [poisson_distribution](#).

Remarks

Precondition: `0.0 < mean`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

random_device Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence from an external device.

Syntax

```
class random_device {
public:
    typedef unsigned int result_type;

    // constructor
    explicit random_device(const std::string& token = "");

    // properties
    static result_type min();
    static result_type max();
    double entropy() const;

    // generate
    result_type operator()();

    // no-copy functions
    random_device(const random_device&) = delete;
    void operator=(const random_device&) = delete;
};
```

Members

random_device	entropy
random_device::operator()	

Remarks

The class describes a source of random numbers, and is allowed but not required to be non-deterministic or cryptographically secure by the ISO C++ Standard. In the Visual Studio implementation the values produced are non-deterministic and cryptographically secure, but runs more slowly than generators created from engines and engine adaptors (such as [mersenne_twister_engine](#), the high quality and fast engine of choice for most applications).

`random_device` results are uniformly distributed in the closed range [`0`, `232`).

`random_device` is not guaranteed to result in a non-blocking call.

Generally, `random_device` is used to seed other generators created with engines or engine adaptors. For more information, see [<random>](#).

Example

The following code demonstrates basic functionality of this class and example results. Because of the non-

deterministic nature of `random_device`, the random values shown in the **Output** section will not match your results. This is normal and expected.

```
// random_device_engine.cpp
// cl.exe /W4 /nologo /EHsc /MTd
#include <random>
#include <iostream>
using namespace std;

int main()
{
    random_device gen;

    cout << "entropy == " << gen.entropy() << endl;
    cout << "min == " << gen.min() << endl;
    cout << "max == " << gen.max() << endl;

    cout << "a random value == " << gen() << endl;
    cout << "a random value == " << gen() << endl;
    cout << "a random value == " << gen() << endl;
}
```

```
entropy == 32
min == 0
max == 4294967295
a random value == 2378414971
a random value == 3633694716
a random value == 213725214
```

This example is simplistic and not representative of the general use-case for this generator. For a more representative code example, see [<random>](#).

Requirements

Header: `<random>`

Namespace: `std`

`random_device::random_device`

Constructs the generator.

```
random_device(const std::string& = "");
```

Remarks

The constructor initializes the generator as needed, ignoring the string parameter. Throws a value of an implementation-defined type derived from [exception](#) if the `random_device` could not be initialized.

`random_device::entropy`

Estimates the randomness of the source.

```
double entropy() const noexcept;
```

Remarks

The member function returns an estimate of the randomness of the source, as measured in bits.

random_device::operator()

Returns a random value.

```
result_type operator()();
```

Remarks

Returns values uniformly distributed in the closed interval [`min`, `max`] as determined by member functions `min()` and `max()`. Throws a value of an implementation-defined type derived from [exception](#) if a random number could not be obtained.

See also

[<random>](#)

seed_seq Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Stores a vector of unsigned integer values that can supply a randomized seed for a random-number engine.

Syntax

```
class seed_seq
{
public:
    // types
    typedef unsigned int result_type;

    // constructors
    seed_seq();
    template <class T>
        seed_seq(initializer_list<T> initlist);
    template <class InputIterator>
        seed_seq(InputIterator begin, InputIterator end);

    // generating functions
    template <class RandomAccessIterator>
        void generate(RandomAccessIterator begin, RandomAccessIterator end);

    // property functions
    size_t size() const;
    template <class OutputIterator>
        void param(OutputIterator dest) const;

    // no copy functions
    seed_seq(const seed_seq&) = delete;
    void operator=(const seed_seq&) = delete;
};
```

Types

```
typedef unsigned int result_type;
```

The type of the elements of the seed sequence. A 32-bit unsigned integer type.

Constructors

```
seed_seq();
```

Default constructor, initializes to have an empty internal sequence.

```
template<class T>
seed_seq(initializer_list<T> initlist);
```

Uses `initlist` to set the internal sequence. `T` must be an integer type.

```
template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);
```

Initializes the internal sequence using all elements in the input iterator range provided.

`iterator_traits<InputIterator>::value_type` must be an integer type.

Members

Generating Functions

```
template<class RandomAccessIterator>
void generate(RandomAccessIterator begin,
             RandomAccessIterator end);
```

Populates the elements of the provided sequence using an internal algorithm. This algorithm is affected by the internal sequence with which `seed_seq` was initialized. Does nothing if `begin == end`.

Property Functions

```
size_t size() const;
```

Returns the number of elements in the `seed_seq`.

```
template<class OutputIterator>
void param(OutputIterator dest) const;
```

Copies the internal sequence into the output iterator `dest`.

Example

The following code example exercises the three constructors and generates output from the resulting `seed_seq` instances when assigned to an array. For an example that uses `seed_seq` with a random number generator, see [<random>](#).

```

#include <iostream>
#include <random>
#include <string>
#include <array>

using namespace std;

void test(const seed_seq& sseq) {
    cout << endl << "seed_seq::size(): " << sseq.size() << endl;

    cout << "seed_seq::param(): ";
    ostream_iterator<unsigned int> out(cout, " ");
    sseq.param(out);
    cout << endl;

    cout << "Generating a sequence of 5 elements into an array: " << endl;
    array<unsigned int, 5> seq;
    sseq.generate(seq.begin(), seq.end());
    for (unsigned x : seq) { cout << x << endl; }
}

int main()
{
    seed_seq seed1;
    test(seed1);

    seed_seq seed2 = { 1701, 1729, 1791 };
    test(seed2);

    string sstr = "A B C D"; // seed string
    seed_seq seed3(sstr.begin(), sstr.end());
    test(seed3);
}

```

```

seed_seq::size(): 0
seed_seq::param():
Generating a sequence of 5 elements into an array:
505382999
163489202
3932644188
763126080
73937346

seed_seq::size(): 3
seed_seq::param(): 1701 1729 1791
Generating a sequence of 5 elements into an array:
1730669648
1954224479
2809786021
1172893117
2393473414

seed_seq::size(): 7
seed_seq::param(): 65 32 66 32 67 32 68
Generating a sequence of 5 elements into an array:
3139879222
3775111734
1084804564
2485037668
1985355432

```

Remarks

Member functions of this class do not throw exceptions.

Requirements

Header: <random>

Namespace: std

See also

[<random>](#)

shuffle_order_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence by reordering the values returned from its base engine.

Syntax

```
template <class Engine, size_t K>
class shuffle_order_engine;
```

Parameters

Engine

The base engine type.

K

Table size. Number of elements in the buffer (table). **Precondition:** $0 < K$

Members

<code>shuffle_order_engine::shuffle_order_engine</code>	<code>shuffle_order_engine::base</code>	<code>shuffle_order_engine::discard</code>
<code>shuffle_order_engine::operator()</code>	<code>shuffle_order_engine::base_type</code>	<code>shuffle_order_engine::seed</code>

For more information about engine members, see [<random>](#).

Remarks

This template class describes an *engine adaptor* that produces values by reordering the values returned by its base engine. Each constructor fills the internal table with *K* values returned by the base engine, and a random element is selected from the table when a value is requested.

Requirements

Header: `<random>`

Namespace: `std`

See also

[<random>](#)

student_t_distribution Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Generates a Student's t -distribution.

Syntax

```
template<class RealType = double>
class student_t_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit student_t_distribution(result_type n = 1.0);
    explicit student_t_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified integral type, or type **double** if none is provided, distributed according to the Student's t -Distribution. The following table links to articles about individual members.

student_t_distribution	<code>student_t_distribution::n</code>	<code>student_t_distribution::param</code>
<code>student_t_distribution::operator()</code>		param_type

The property function `n()` returns the value for the stored distribution parameter `n`.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the Student's t -distribution, see the Wolfram MathWorld article [Students \$t\$ -Distribution](#).

Example

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double n, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::student_t_distribution<> distr(n);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double n_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'n' distribution parameter (must be greater than zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(n_dist, samples);
}
```

```
Use CTRL-Z to bypass data entry and run using default values.  
Enter a floating point value for the 'n' distribution parameter (must be greater than zero): 1  
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308  
max() == 1.79769e+308  
n() == 1.0000000000  
Distribution for 10 samples:  
1: -1.3084956212  
2: -1.0899518684  
3: -0.9568771388  
4: -0.9372088821  
5: -0.7381334669  
6: -0.2488074854  
7: -0.2028714601  
8: 1.4013074495  
9: 5.3244792236  
10: 92.7084335614
```

Requirements

Header: <random>

Namespace: std

student_t_distribution::student_t_distribution

Constructs the distribution.

```
explicit student_t_distribution(RealType n = 1.0);  
explicit student_t_distribution(const param_type& parm);
```

Parameters

n

The `n` distribution parameter.

parm

The parameter package used to construct the distribution.

Remarks

Precondition: $0.0 < n$

The first constructor constructs an object whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

student_t_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef student_t_distribution<result_type> distribution_type;
    param_type(result_type n = 1.0);
    result_type n() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

n

The `n` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < n`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

subtract_with_carry_engine Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Generates a random sequence by the subtract-with-carry (lagged Fibonacci) algorithm.

Syntax

```
template <class UIntType, size_t W, size_t S, size_t R>
class subtract_with_carry_engine;
```

Parameters

UIntType

The unsigned integer result type. For possible types, see [<random>](#).

W

Word size. Size of each word, in bits, of the state sequence. **Precondition:**

$0 < W \leq \text{numeric_limits}\langle \text{UIntType} \rangle::\text{digits}$

S

Short lag. Number of integer values. **Precondition:** $0 < S < R$

R

Long lag. Determines recurrence in the series generated.

Members

<code>subtract_with_carry_engine::subtract_w</code>	<code>subtract_with_carry_engine::min</code>	<code>subtract_with_carry_engine::discard</code>
<code>subtract_with_carry_engine::operator()</code>	<code>subtract_with_carry_engine::max</code>	<code>subtract_with_carry_engine::seed</code>
<code>default_seed</code> is a member constant, defined as <code>19780503u</code> , used as the default parameter value for <code>subtract_with_carry_engine::seed</code> and the single value constructor.		

For more information about engine members, see [<random>](#).

Remarks

The `subtract_with_carry_engine` template class is an improvement over the [linear_congruential_engine](#). Neither for these engines is as fast or with as high quality results as the [mersenne_twister_engine](#).

This engine produces values of a user-specified unsigned integral type using the recurrence relation (*period*)

$x(i) = (x(i - R) - x(i - S) - cy(i - 1)) \bmod M$, where `cy(i)` has the value `1` if $x(i - S) - x(i - R) - cy(i - 1) < 0$, otherwise `0`, and `M` has the value 2^W . The engine's state is a carry indicator plus *R* values. These values consist of the last *R* values returned if `operator()` has been called at least *R* times, otherwise the `N` values that have been returned and the last `R - N` values of the seed.

The template argument `UIntType` must be large enough to hold values up to `M - 1`.

Although you can construct a generator from this engine directly, you can also use one of these predefined typedefs:

`ranlux24_base` : Used as a base for `ranlux24` .

```
typedef subtract_with_carry_engine<unsigned int, 24, 10, 24> ranlux24_base;
```

`ranlux48_base` : Used as a base for `ranlux48` .

```
typedef subtract_with_carry_engine<unsigned long long, 48, 5, 12> ranlux48_base;
```

For detailed information about the subtract with carry engine algorithm, see the Wikipedia article [Lagged Fibonacci generator](#).

Requirements

Header: `<random>`

Namespace: `std`

See also

[<random>](#)

uniform_int_distribution Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Generates a uniform (every value is equally probable) integer distribution within an output range that is inclusive-inclusive.

Syntax

```
template<class IntType = int>
    class uniform_int_distribution {
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructors and reset functions
    explicit uniform_int_distribution(
        result_type a = 0, result_type b = numeric_limits<result_type>::max());
    explicit uniform_int_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
        result_type operator()(URNG& gen);
    template <class URNG>
        result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to **int**. For possible types, see [<random>](#).

Remarks

The template class describes an inclusive-inclusive distribution that produces values of a user-specified integral type with a distribution so that every value is equally probable. The following table links to articles about individual members.

uniform_int_distribution	<code>uniform_int_distribution::a</code>	<code>uniform_int_distribution::param</code>
<code>uniform_int_distribution::operator()</code>	<code>uniform_int_distribution::b</code>	param_type

The property member `a()` returns the currently stored minimum bound of the distribution, while `b()` returns the currently stored maximum bound. For this distribution class, these minimum and maximum values are the same as those returned by the common property functions `min()` and `max()`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int a, const int b, const int s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::uniform_int_distribution<> distr(a, b);

    std::cout << "lower bound == " << distr.a() << std::endl;
    std::cout << "upper bound == " << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int a_dist = 1;
    int b_dist = 10;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter an integer value for the lower bound of the distribution: ";
    std::cin >> a_dist;
    std::cout << "Enter an integer value for the upper bound of the distribution: ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}

```



```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for the lower bound of the distribution: 0
Enter an integer value for the upper bound of the distribution: 12
Enter an integer value for the sample count: 200
lower bound == 0
upper bound == 12
Distribution for 200 samples:
 0 :::::::::::::::
 1 :::::::::::::::
 2 :::::::::::::::
 3 :::::::::::::::
 4 :::::::::::::::
 5 :::::::::::::::
 6 :::::::::::::::
 7 :::::::::::::::
 8 :::::::::::::::
 9 :::::::::::::::
10 :::::::::::::::
11 :::::::::::::::
12 :::::::::::::::
```

Requirements

Header: <random>

Namespace: std

uniform_int_distribution::uniform_int_distribution

Constructs the distribution.

```
explicit uniform_int_distribution(
    result_type a = 0, result_type b = std::numeric_limits<result_type>::max());
explicit uniform_int_distribution(const param_type& parm);
```

Parameters

a

The lower bound for random values, inclusive.

b

The upper bound for random values, inclusive.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $a \leq b$

The first constructor constructs an object whose stored *a* value holds the value *a* and whose stored *b* value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

uniform_int_distribution::param_type

Stores the parameters of the distribution.

```

struct param_type {
    typedef uniform_int_distribution<result_type> distribution_type;
    param_type(
        result_type a = 0, result_type b = std::numeric_limits<result_type>::max());
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};

```

Parameters

a

The lower bound for random values, inclusive.

b

The upper bound for random values, inclusive.

right

The `param_type` object to compare to this.

Remarks

Precondition: `a ≤ b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

uniform_real_distribution Class

11/9/2018 • 4 minutes to read • [Edit Online](#)

Generates a uniform (every value is equally probable) floating-point distribution within an output range that is inclusive-exclusive.

Syntax

```
template<class RealType = double>
    class uniform_real_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructors and reset functions
    explicit uniform_real_distribution(
        result_type a = 0.0, result_type b = 1.0);
    explicit uniform_real_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
        result_type operator()(URNG& gen);
    template <class URNG>
        result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

The template class describes an inclusive-exclusive distribution that produces values of a user-specified integral floating point type with a distribution so that every value is equally probable. The following table links to articles about individual members.

uniform_real_distribution	<code>uniform_real_distribution::a</code>	<code>uniform_real_distribution::param</code>
<code>uniform_real_distribution::operator()</code>	<code>uniform_real_distribution::b</code>	param_type

The property member `a()` returns the currently stored minimum bound of the distribution, while `b()` returns the currently stored maximum bound. For this distribution class, these minimum and maximum values are the same as those returned by the common property functions `min()` and `max()` described in the [<random>](#) topic.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic seed
    // std::random_device rd;
    // std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::uniform_real_distribution<> distr(a,b);

    std::cout << "lower bound == " << distr.a() << std::endl;
    std::cout << "upper bound == " << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 1.0;
    double b_dist = 1.5;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the lower bound of the distribution: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the upper bound of the distribution: ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}

```

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the lower bound of the distribution: 0
Enter a floating point value for the upper bound of the distribution: 1
Enter an integer value for the sample count: 10
lower bound == 0
upper bound == 1
Distribution for 10 samples:
  1: 0.0288609485
  2: 0.2007994386
  3: 0.3027480117
  4: 0.4124758695
  5: 0.4413777133
  6: 0.4846447405
  7: 0.6225745916
  8: 0.6880935217
  9: 0.7541936723
 10: 0.8795716566
```

Requirements

Header: <random>

Namespace: std

uniform_real_distribution::uniform_real_distribution

Constructs the distribution.

```
explicit uniform_real_distribution(result_type a = 0.0, result_type b = 1.0);
explicit uniform_real_distribution(const param_type& parm);
```

Parameters

a

The lower bound for random values, inclusive.

b

The upper bound for random values, exclusive.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: `a < b`

The first constructor constructs an object whose stored *a* value holds the value *a* and whose stored *b* value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

uniform_real_distribution::param_type

Stores all the parameters of the distribution.

```
struct param_type {
    typedef uniform_real_distribution<result_type> distribution_type;
    param_type(result_type a = 0.0, result_type b = 1.0);
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

a

The lower bound for random values, inclusive.

b

The upper bound for random values, exclusive.

right

The `param_type` object to compare to this.

Remarks

Precondition: `a < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

weibull_distribution Class

11/9/2018 • 4 minutes to read • [Edit Online](#)

Generates a Weibull distribution.

Syntax

```
class weibull_distribution
{
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit weibull_distribution(result_type a = 1.0, result_type b = 1.0);
    explicit weibull_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
        result_type operator()(URNG& gen);
    template <class URNG>
        result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to **double**. For possible types, see [<random>](#).

Remarks

The template class describes a distribution that produces values of a user-specified floating point type, or type **double** if none is provided, distributed according to the Weibull Distribution. The following table links to articles about individual members.

weibull_distribution	<code>weibull_distribution::a</code>	<code>weibull_distribution::param</code>
<code>weibull_distribution::operator()</code>	<code>weibull_distribution::b</code>	param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters *a* and *b*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result,

respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see [<random>](#).

For detailed information about the Weibull distribution, see the Wolfram MathWorld article [Weibull Distribution](#).

Example

```

// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    // std::random_device gen;
    std::mt19937 gen(1701);

    std::weibull_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) << std::setprecision(10) << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter a floating point value for the 'a' distribution parameter (must be greater than zero):";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'b' distribution parameter (must be greater than zero):";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}

```

Output

First run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter (must be greater than zero): 1
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == 0
max() == 1.79769e+308
a() == 1.0000000000
b() == 1.0000000000
Distribution for 10 samples:
1: 0.0936880533
2: 0.1225944894
3: 0.6443593183
4: 0.6551171649
5: 0.7313457551
6: 0.7313557977
7: 0.7590097389
8: 1.4466885214
9: 1.6434088411
10: 2.1201210996
```

Second run:

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter (must be greater than zero): .5
Enter a floating point value for the 'b' distribution parameter (must be greater than zero): 5.5
Enter an integer value for the sample count: 10
```

```
min() == 0
max() == 1.79769e+308
a() == 0.5000000000
b() == 5.5000000000
Distribution for 10 samples:
1: 0.0482759823
2: 0.0826617486
3: 2.2835941207
4: 2.3604817485
5: 2.9417663742
6: 2.9418471657
7: 3.1685268104
8: 11.5109922290
9: 14.8543594043
10: 24.7220241239
```

Requirements

Header: <random>

Namespace: std

weibull_distribution::weibull_distribution

```
explicit weibull_distribution(result_type a = 1.0, result_type b = 1.0);
explicit weibull_distribution(const param_type& parm);
```

Parameters

a

The `a` distribution parameter.

b

The `b` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: $0.0 < a$ and $0.0 < b$

The first constructor constructs an object whose stored `a` value holds the value a and whose stored `b` value holds the value b .

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

weibull_distribution::param_type

Stores the parameters of the distribution.

```
struct param_type {
    typedef weibull_distribution<result_type> distribution_type;
    param_type(result_type a = 1.0, result_type b = 1.0);
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
};
```

Parameters

a

The `a` distribution parameter.

b

The `b` distribution parameter.

right

The `param_type` object to compare to this.

Remarks

Precondition: $0.0 < a$ and $0.0 < b$

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

[<random>](#)

<ratio>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Include the standard header <ratio> to define constants and templates that are used to store and manipulate rational numbers at compile time.

Syntax

```
#include <ratio>
```

ratio Template

```
template<std::intmax_t Numerator, std::intmax_t Denominator = 1>
struct ratio // holds the ratio of Numerator to Denominator
{
    static constexpr std::intmax_t num;
    static constexpr std::intmax_t den;
    typedef ratio<num, den> type;
}
```

The template `ratio` defines the static constants `num` and `den` such that `num / den == Numerator / Denominator` and `num` and `den` have no common factors. `num / den` is the value that is represented by the template class. Therefore, `type` designates the instantiation `ratio<num, den>`.

Specializations

<ratio> also defines specializations of `ratio` that have the following form.

```
template <class R1, class R2> struct ratio_specialization
```

Each specialization takes two template parameters that must also be specializations of `ratio`. The value of `type` is determined by an associated logical operation.

NAME	TYPE VALUE
<code>ratio_add</code>	<code>R1 + R2</code>
<code>ratio_divide</code>	<code>R1 / R2</code>
<code>ratio_equal</code>	<code>R1 == R2</code>
<code>ratio_greater</code>	<code>R1 > R2</code>
<code>ratio_greater_equal</code>	<code>R1 >= R2</code>
<code>ratio_less</code>	<code>R1 < R2</code>
<code>ratio_less_equal</code>	<code>R1 <= R2</code>
<code>ratio_multiply</code>	<code>R1 * R2</code>

NAME	TYPEVALUE
ratio_not_equal	!(R1 == R2)
ratio_subtract	R1 - R2

typedefs

For convenience, the header defines ratios for the standard SI prefixes:

```
typedef ratio<1, 1000000000000000> atto;
typedef ratio<1, 100000000000000> femto;
typedef ratio<1, 100000000000> pico;
typedef ratio<1, 100000000> nano;
typedef ratio<1, 100000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000, 1> tera;
typedef ratio<1000000000000000, 1> peta;
typedef ratio<1000000000000000000, 1> exa;
```

See also

[Header Files Reference](#)

<regex>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a template class to parse [Regular Expressions \(C++\)](#), and several template classes and functions to search text for matches to a regular expression object.

Syntax

```
#include <regex>
```

Remarks

To create a regular expression object, use the template class [basic_regex Class](#) or one of its specializations, [regex](#) and [wregex](#), together with the syntax flags of type [regex_constants::syntax_option_type](#).

To search text for matches to a regular expression object, use the template functions [regex_match](#) and [regex_search](#), together with the match flags of type [regex_constants::match_flag_type](#). These functions return results by using the template class [match_results Class](#) and its specializations, [cmatch](#), [wcmatch](#), [smatch](#), and [wsmatch](#), together with the template class [sub_match Class](#) and its specializations, [csub_match](#), [wsub_match](#), [ssub_match](#), and [wssub_match](#).

To replace text that matches a regular expression object, use the template function [regex_replace](#), together with the match flags of type [regex_constants::match_flag_type](#).

To iterate through multiple matches of a regular expression object, use the template classes [regex_iterator Class](#) and [regex_token_iterator Class](#) or one of their specializations, [cregex_iterator](#), [sregex_iterator](#), [wcregex_iterator](#), [wsregex_iterator](#), [cregex_token_iterator](#), [sregex_token_iterator](#), [wcregex_token_iterator](#), or [wsregex_token_iterator](#), together with the match flags of type [regex_constants::match_flag_type](#).

To modify the details of the grammar of regular expressions, write a class that implements the regular expression traits.

Classes

CLASS	DESCRIPTION
basic_regex	Wraps a regular expression.
match_results	Holds a sequence of submatches.
regex_constants	Holds assorted constants.
regex_error	Reports a bad regular expression.
regex_iterator	Iterates through match results.
regex_traits	Describes characteristics of elements for matching.
regex_traits<char>	Describes characteristics of char for matching.

CLASS	DESCRIPTION
<code>regex_traits<wchar_t></code>	Describes characteristics of wchar_t for matching.
<code>regex_token_iterator</code>	Iterates through submatches.
<code>sub_match</code>	Describes a submatch.

Type Definitions

<code>cmatch</code>	Type definition for char <code>match_results</code> .
<code>cregex_iterator</code>	Type definition for char <code>regex_iterator</code> .
<code>cregex_token_iterator</code>	Type definition for char <code>regex_token_iterator</code> .
<code>csub_match</code>	Type definition for char <code>sub_match</code> .
<code>regex</code>	Type definition for char <code>basic_regex</code> .
<code>smatch</code>	Type definition for <code>string</code> <code>match_results</code> .
<code>sregex_iterator</code>	Type definition for <code>string</code> <code>regex_iterator</code> .
<code>sregex_token_iterator</code>	Type definition for <code>string</code> <code>regex_token_iterator</code> .
<code>ssub_match</code>	Type definition for <code>string</code> <code>sub_match</code> .
<code>wcmatch</code>	Type definition for wchar_t <code>match_results</code> .
<code>wcregex_iterator</code>	Type definition for wchar_t <code>regex_iterator</code> .
<code>wcregex_token_iterator</code>	Type definition for wchar_t <code>regex_token_iterator</code> .
<code>wcsub_match</code>	Type definition for wchar_t <code>sub_match</code> .
<code>wregex</code>	Type definition for wchar_t <code>basic_regex</code> .
<code>wsmatch</code>	Type definition for <code>wstring</code> <code>match_results</code> .
<code>wsregex_iterator</code>	Type definition for <code>wstring</code> <code>regex_iterator</code> .
<code>wsregex_token_iterator</code>	Type definition for <code>wstring</code> <code>regex_token_iterator</code> .
<code>wssub_match</code>	Type definition for <code>wstring</code> <code>sub_match</code> .

Functions

FUNCTION	DESCRIPTION
regex_match	Exactly matches a regular expression.
regex_replace	Replaces matched regular expressions.
regex_search	Searches for a regular expression match.
swap	Swaps <code>basic_regex</code> or <code>match_results</code> objects.

Operators

OPERATOR	DESCRIPTION
operator==	Comparison of various objects, equal.
operator!=	Comparison of various objects, not equal.
operator<	Comparison of various objects, less than.
operator<=	Comparison of various objects, less than or equal.
operator>	Comparison of various objects, greater than.
operator>=	Comparison of various objects, greater than or equal.
operator<<	Inserts a <code>sub_match</code> in a stream.

See also

[Regular Expressions \(C++\)](#)

[regex_constants](#) Class

[regex_error](#) Class

[<regex>](#) functions

[regex_iterator](#) Class

[<regex>](#) operators

[regex_token_iterator](#) Class

[regex_traits](#) Class

[<regex>](#) typedefs

<regex> functions

10/31/2018 • 8 minutes to read • [Edit Online](#)

<code>regex_match</code>	Tests whether a regular expression matches the entire target string.
<code>regex_replace</code>	Replaces matched regular expressions.
<code>regex_search</code>	Searches for a regular expression match.
<code>swap</code>	Swaps two <code>basic_regex</code> or <code>match_results</code> objects.

regex_match

Tests whether a regular expression matches the entire target string.

```

// (1)
template <class BidIt, class Alloc, class Elem, class RXtraits, class Alloc2>
bool regex_match(
    BidIt first,
    BidIt last,
    match_results<BidIt, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (2)
template <class BidIt, class Elem, class RXtraits, class Alloc2>
bool regex_match(
    BidIt first,
    BidIt last,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (3)
template <class Elem, class Alloc, class RXtraits, class Alloc2>
bool regex_match(
    const Elem *ptr,
    match_results<const Elem*, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (4)
template <class Elem, class RXtraits, class Alloc2>
bool regex_match(
    const Elem *ptr,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (5)
template <class IOtraits, class IOalloc, class Alloc, class Elem, class RXtraits, class Alloc2>
bool regex_match(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    match_results<typename basic_string<Elem, IOtraits, IOalloc>::const_iterator, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (6)
template <class IOtraits, class IOalloc, class Elem, class RXtraits, class Alloc2>
bool regex_match(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

```

Parameters

BidIt

The iterator type for submatches. For common cases this one of `string::const_iterator`, `wstring::const_iterator`, `const char*` or `const wchar_t*`.

Alloc

The match results allocator class.

Elem

The type of elements to match. For common cases this is `string`, `wstring`, `char*` or `wchar_t*`.

RXtraits

Traits class for elements.

Alloc2

The regular expression allocator class.

IOtraits

The string traits class.

IOalloc

The string allocator class.

flags

Flags for matches.

first

Beginning of sequence to match.

last

End of sequence to match.

match

The match results. Corresponds to Elem type: [smatch](#) for `string`, [wsmatch](#) for `wstring`, [cmatch](#) for `char*` or [wcmatch](#) for `wchar_t*`.

ptr

Pointer to beginning of sequence to match. If *ptr* is `char*`, then use `cmatch` and `regex`. If *ptr* is `wchar_t*` then use `wcmatch` and `wregex`.

re

The regular expression to match. Type `regex` for `string` and `char*`, or `wregex` for `wstring` and `wchar_t*`.

str

String to match. Corresponds to the type of *Elem*.

Remarks

Each template function returns true only if the entire operand sequence *str* exactly matches the regular expression argument *re*. Use [regex_search](#) to match a substring within a target sequence and `regex_iterator` to find multiple matches. The functions that take a `match_results` object set its members to reflect whether the match succeeded and if so what the various capture groups in the regular expression captured.

The functions that take a `match_results` object set its members to reflect whether the match succeeded and if so what the various capture groups in the regular expression captured.

Example

```

// std__regex__regex_match.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

using namespace std;

int main()
{
    // (1) with char*
    // Note how const char* requires cmatch and regex
    const char *first = "abc";
    const char *last = first + strlen(first);
    cmatch narrowMatch;
    regex rx("a(b)c");

    bool found = regex_match(first, last, narrowMatch, rx);
    if (found)
        wcout << L"Regex found in abc" << endl;

    // (2) with std::wstring
    // Note how wstring requires wsmatch and wregex.
    // Note use of const iterators cbegin() and cend().
    wstring target(L"Hello");
    wsmatch wideMatch;
    wregex wrx(L"He(l+)o");

    if (regex_match(target.cbegin(), target.cend(), wideMatch, wrx))
        wcout << L"The matching text is:" << wideMatch.str() << endl;

    // (3) with std::string
    string target2("Drizzle");
    regex rx2(R"(D\w+e)"); // no double backslashes with raw string literal

    found = regex_match(target2.cbegin(), target2.cend(), rx2);
    if (found)
        wcout << L"Regex found in Drizzle" << endl;

    // (4) with wchar_t*
    const wchar_t* target3 = L"2014-04-02";
    wcmatch wideMatch2;

    // LR"(...)" is a raw wide-string literal. Open and close parens
    // are delimiters, not string elements.
    wregex wrx2(LR"(\d{4}(-|/)\d{2}(-|/)\d{2})");
    if (regex_match(target3, wideMatch2, wrx2))
    {
        wcout << L"Matching text: " << wideMatch2.str() << endl;
    }

    return 0;
}

```

```

Regex found in abc
The matching text is: Hello
Regex found in Drizzle
The matching text is: 2014-04-02

```

regex_replace

Replaces matched regular expressions.

```

template <class OutIt, class BidIt, class RXtraits, class Alloc, class Elem>
OutIt regex_replace(
    OutIt out,
    BidIt first,
    BidIt last,
    const basic_regex<Elem, RXtraits, Alloc>& re,
    const basic_string<Elem>& fmt,
    match_flag_type flags = match_default);

template <class RXtraits, class Alloc, class Elem>
basic_string<Elem> regex_replace(
    const basic_string<Elem>& str,
    const basic_regex<Elem, RXtraits, Alloc>& re,
    const basic_string<Elem>& fmt,
    match_flag_type flags = match_default);

```

Parameters

OutIt

The iterator type for replacements.

BidIt

The iterator type for submatches.

RXtraits

Traits class for elements.

Alloc

The regular expression allocator class.

Elem

The type of elements to match.

flags

Flags for matches.

first

Beginning of sequence to match.

fmt

The format for replacements.

last

End of sequence to match.

out

The output iterator.

re

The regular expression to match.

str

String to match.

Remarks

The first function constructs a [regex_iterator Class](#) object `iter(first, last, re, flags)` and uses it to split its input range `[first, last)` into a series of subsequences `T0 M0 T1 M1...TN-1 MN-1 TN`, where `Mn` is the *n*th match detected by the iterator. If no matches are found, `T0` is the entire input range and `N` is zero. If `(flags & format_first_only) != 0` only the first match is used, `T1` is all of the input text that follows the match, and `N` is 1. For each `i` in the range `[0, N)`, if `(flags & format_no_copy) == 0` it copies the text in the range `Ti`

to the iterator `out`. It then calls `m.format(out, fmt, flags)`, where `m` is the `match_results` object returned by the iterator object `iter` for the subsequence `Mi`. Finally, if `(flags & format_no_copy) == 0` it copies the text in the range `TN` to the iterator `out`. The function returns `out`.

The second function constructs a local variable `result` of type `basic_string<charT>` and calls `regex_replace(back_inserter(result), str.begin(), str.end(), re, fmt, flags)`. It returns `result`.

Example

```
// std__regex__regex_replace.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    char buf[20];
    const char *first = "axayaz";
    const char *last = first + strlen(first);
    std::regex rx("a");
    std::string fmt("A");
    std::regex_constants::match_flag_type fonly =
        std::regex_constants::format_first_only;

    *std::regex_replace(&buf[0], first, last, rx, fmt) = '\0';
    std::cout << "replacement == " << &buf[0] << std::endl;

    *std::regex_replace(&buf[0], first, last, rx, fmt, fonly) = '\0';
    std::cout << "replacement == " << &buf[0] << std::endl;

    std::string str("adaeaf");
    std::cout << "replacement == "
        << std::regex_replace(str, rx, fmt) << std::endl;

    std::cout << "replacement == "
        << std::regex_replace(str, rx, fmt, fonly) << std::endl;

    return (0);
}
```

```
replacement == AxAyAz
replacement == Axayaz
replacement == AdAeAf
replacement == Adaeaf
```

regex_search

Searches for a regular expression match.

```

template <class BidIt, class Alloc, class Elem, class RXtraits, class Alloc2>
bool regex_search(
    BidIt first,
    Bidit last,
    match_results<BidIt, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class BidIt, class Elem, class RXtraits, class Alloc2>
bool regex_search(
    BidIt first,
    Bidit last,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class Elem, class Alloc, class RXtraits, class Alloc2>
bool regex_search(
    const Elem* ptr,
    match_results<const Elem*, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class Elem, class RXtraits, class Alloc2>
bool regex_search(
    const Elem* ptr,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class IOtraits, class IOalloc, class Alloc, class Elem, class RXtraits, class Alloc2>
bool regex_search(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    match_results<typename basic_string<Elem, IOtraits, IOalloc>::const_iterator, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class IOtraits, class IOalloc, class Elem, class RXtraits, class Alloc2>
bool regex_search(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

```

Parameters

BidIt

The iterator type for submatches.

Alloc

The match results allocator class.

Elem

The type of elements to match.

RXtraits

Traits class for elements.

Alloc2

The regular expression allocator class.

IOtraits

The string traits class.

IOalloc

The string allocator class.

flags

Flags for matches.

first

Beginning of sequence to match.

last

End of sequence to match.

match

The match results.

ptr

Pointer to beginning of sequence to match.

re

The regular expression to match.

str

String to match.

Remarks

Each template function returns true only if a search for its regular expression argument *re* in its operand sequence succeeds. The functions that take a `match_results` object set its members to reflect whether the search succeeded and if so what the various capture groups in the regular expression captured.

Example

```

// std__regex__regex_search.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    const char *first = "abcd";
    const char *last = first + strlen(first);
    std::cmatch mr;
    std::regex rx("abc");
    std::regex_constants::match_flag_type fl =
        std::regex_constants::match_default;

    std::cout << "search(f, f+1, \"abc\") == " << std::boolalpha
        << regex_search(first, first + 1, rx, fl) << std::endl;

    std::cout << "search(f, l, \"abc\") == " << std::boolalpha
        << regex_search(first, last, mr, rx) << std::endl;
    std::cout << "    matched: \"" << mr.str() << "\"" << std::endl;

    std::cout << "search(\"a\", \"abc\") == " << std::boolalpha
        << regex_search("a", rx) << std::endl;

    std::cout << "search(\"xabcd\", \"abc\") == " << std::boolalpha
        << regex_search("xabcd", mr, rx) << std::endl;
    std::cout << "    matched: \"" << mr.str() << "\"" << std::endl;

    std::cout << "search(string, \"abc\") == " << std::boolalpha
        << regex_search(std::string("a"), rx) << std::endl;

    std::string str("abcabc");
    std::match_results<std::string::const_iterator> mr2;
    std::cout << "search(string, \"abc\") == " << std::boolalpha
        << regex_search(str, mr2, rx) << std::endl;
    std::cout << "    matched: \"" << mr2.str() << "\"" << std::endl;

    return (0);
}

```

```

search(f, f+1, "abc") == false
search(f, l, "abc") == true
    matched: "abc"
search("a", "abc") == false
search("xabcd", "abc") == true
    matched: "abc"
search(string, "abc") == false
search(string, "abc") == true
    matched: "abc"

```

swap

Swaps two `basic_regex` or `match_results` objects.

```

template <class Elem, class RXtraits>
void swap(
    basic_regex<Elem, RXtraits, Alloc>& left,
    basic_regex<Elem, RXtraits>& right) noexcept;

template <class Elem, class IOtraits, class BidIt, class Alloc>
void swap(
    match_results<BidIt, Alloc>& left,
    match_results<BidIt, Alloc>& right) noexcept;

```

Parameters

Elem

The type of elements to match.

RXtraits

Traits class for elements.

Remarks

The template functions swap the contents of their respective arguments in constant time and do not throw exceptions.

Example

```

// std__regex__swap.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx0("c(a*)|(b)");
    std::regex rx1;
    std::cmatch mr0;
    std::cmatch mr1;

    swap(rx0, rx1);
    std::regex_search("xcaaay", mr1, rx1);
    swap(mr0, mr1);

    std::csub_match sub = mr0[1];
    std::cout << "matched == " << std::boolalpha
        << sub.matched << std::endl;
    std::cout << "length == " << sub.length() << std::endl;
    std::cout << "string == " << sub << std::endl;

    return (0);
}

```

```

matched == true
length == 3
string == aaa

```

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_token_iterator Class](#)

[regex_traits Class](#)

[<regex> typedefs](#)

<regex> operators

11/9/2018 • 11 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<<</code>	<code>operator<=</code>
<code>operator==</code>		

operator!=

Not equal comparison for various objects.

```
template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator!=(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator!=(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator!=(const typename iterator_traits<BidIt>::value_type *left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator!=(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

template <class BidIt, class Alloc>
bool operator!=(const match_results<BidIt, Alloc>& left,
               const match_results<BidIt, Alloc>& right);
```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `!(left == right)`.

Example

```
// std__regex__operator_ne.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "match == " << mr.str() << std::endl;
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "match != match == " << std::boolalpha
        << (mr != mr) << std::endl;
    std::cout << "sub != sub == " << std::boolalpha
        << (sub != sub) << std::endl;

    std::cout << "string(\"aab\") != sub == " << std::boolalpha
        << (Mystr("aab") != sub) << std::endl;
    std::cout << "sub != string(\"aab\") == " << std::boolalpha
        << (sub != Mystr("aab")) << std::endl;

    std::cout << "\"aab\" != sub == " << std::boolalpha
        << ("aab" != sub) << std::endl;
    std::cout << "sub != \"aab\" == " << std::boolalpha
        << (sub != "aab") << std::endl;

    std::cout << "'a' != sub == " << std::boolalpha
        << ('a' != sub) << std::endl;
    std::cout << "sub != 'a' == " << std::boolalpha
        << (sub != 'a') << std::endl;

    return (0);
}
```

```

match == caaa
sub == aaa

match != match == false
sub != sub == false
string("aab") != sub == true
sub != string("aab") == true
"aab" != sub == true
sub != "aab" == true
'a' != sub == true
sub != 'a' == true

```

operator<

Less than comparison for various objects.

```

template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator<(const typename iterator_traits<BidIt>::value_type *left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator<(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator converts its arguments to a string type and returns true only if the converted value of *left* compares less than the converted value of *right*.

Example

```
// std_regex_operator_lt.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub < sub == " << std::boolalpha
        << (sub < sub) << std::endl;

    std::cout << "string(\"aab\") < sub == " << std::boolalpha
        << (Mystr("aab") < sub) << std::endl;
    std::cout << "sub < string(\"aab\") == " << std::boolalpha
        << (sub < Mystr("aab")) << std::endl;

    std::cout << "\"aab\" < sub == " << std::boolalpha
        << ("aab" < sub) << std::endl;
    std::cout << "sub < \"aab\" == " << std::boolalpha
        << (sub < "aab") << std::endl;

    std::cout << "'a' < sub == " << std::boolalpha
        << ('a' < sub) << std::endl;
    std::cout << "sub < 'a' == " << std::boolalpha
        << (sub < 'a') << std::endl;

    return (0);
}
```

```
sub == aaa

sub < sub == false
string("aab") < sub == false
sub < string("aab") == true
"aab" < sub == false
sub < "aab" == true
'a' < sub == true
sub < 'a' == false
```

operator<<

Inserts a sub_match in a stream.

```
template <class Elem, class IOtraits, class Alloc, class BidIt>
basic_ostream<Elem, IOtraits>& operator<<(basic_ostream<Elem, IOtraits>& os,
    const sub_match<BidIt>& right);
```


Parameters

Elem

The element type.

IOtraits

The string traits class.

Alloc

The allocator class.

BidIt

The iterator type.

os

The output stream.

right

The object to insert.

Remarks

The template operator returns `os << right.str()`.

Example

```
// std__regex_operator_ins.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[0];
    std::cout << "whole match: " << sub << std::endl;

    return (0);
}
```

```
whole match: caaa
```

operator<=

Less than or equal comparison for various objects.

```

template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<=(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<=(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator<=(const typename iterator_traits<BidIt>::value_type *left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator<=(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `!(right < left)`.

Example

```

// std__regex__operator_le.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub <= sub == " << std::boolalpha
        << (sub <= sub) << std::endl;

    std::cout << "string(\"aab\") <= sub == " << std::boolalpha
        << (Mystr("aab") <= sub) << std::endl;
    std::cout << "sub <= string(\"aab\") == " << std::boolalpha
        << (sub <= Mystr("aab")) << std::endl;

    std::cout << "\"aab\" <= sub == " << std::boolalpha
        << ("aab" <= sub) << std::endl;
    std::cout << "sub <= \"aab\" == " << std::boolalpha
        << (sub <= "aab") << std::endl;

    std::cout << "'a' <= sub == " << std::boolalpha
        << ('a' <= sub) << std::endl;
    std::cout << "sub <= 'a' == " << std::boolalpha
        << (sub <= 'a') << std::endl;

    return (0);
}

```

```

sub == aaa

sub <= sub == true
string("aab") <= sub == false
sub <= string("aab") == true
"aab" <= sub == false
sub <= "aab" == true
'a' <= sub == true
sub <= 'a' == false

```

operator==

Equal comparison for various objects.

```

template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator==(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator==(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator==(const typename iterator_traits<BidIt>::value_type* left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type* right);

template <class BidIt>
bool operator==(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

template <class BidIt, class Alloc>
bool operator==(const match_results<BidIt, Alloc>& left,
               const match_results<BidIt, Alloc>& right);

```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator converts each of its arguments to a string type and returns the result of comparing the converted objects for equality.

When a template operator converts its arguments to a string type it uses the first of the following transformations that applies:

arguments whose types are a specialization of template class `match_results` or `sub_match` are converted by calling the `str` member function;

arguments whose types are a specialization of the template class `basic_string` are unchanged;

all other argument types are converted by passing the argument value to the constructor for an appropriate

specialization of the template class `basic_string` .

Example

```
// std__regex__operator_eq.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "match == " << mr.str() << std::endl;
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "match == match == " << std::boolalpha
        << (mr == mr) << std::endl;
    std::cout << "sub == sub == " << std::boolalpha
        << (sub == sub) << std::endl;

    std::cout << "string(\"aab\") == sub == " << std::boolalpha
        << (Mystr("aab") == sub) << std::endl;
    std::cout << "sub == string(\"aab\") == " << std::boolalpha
        << (sub == Mystr("aab")) << std::endl;

    std::cout << "\"aab\" == sub == " << std::boolalpha
        << ("aab" == sub) << std::endl;
    std::cout << "sub == \"aab\" == " << std::boolalpha
        << (sub == "aab") << std::endl;

    std::cout << "'a' == sub == " << std::boolalpha
        << ('a' == sub) << std::endl;
    std::cout << "sub == 'a' == " << std::boolalpha
        << (sub == 'a') << std::endl;

    return (0);
}
```

```
match == caaa
sub == aaa

match == match == true
sub == sub == true
string("aab") == sub == false
sub == string("aab") == false
"aab" == sub == false
sub == "aab" == false
'a' == sub == false
sub == 'a' == false
```

operator>

Greater than comparison for various objects.

```

template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator>(const typename iterator_traits<BidIt>::value_type *left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator>(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `right < left`.

Example

```

// std_regex_operator_gt.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub > sub == " << std::boolalpha
        << (sub > sub) << std::endl;

    std::cout << "string(\"aab\") > sub == " << std::boolalpha
        << (Mystr("aab") > sub) << std::endl;
    std::cout << "sub > string(\"aab\") == " << std::boolalpha
        << (sub > Mystr("aab")) << std::endl;

    std::cout << "\"aab\" > sub == " << std::boolalpha
        << ("aab" > sub) << std::endl;
    std::cout << "sub > \"aab\" == " << std::boolalpha
        << (sub > "aab") << std::endl;

    std::cout << "'a' > sub == " << std::boolalpha
        << ('a' > sub) << std::endl;
    std::cout << "sub > 'a' == " << std::boolalpha
        << (sub > 'a') << std::endl;

    return (0);
}

```

```

sub == aaa

sub > sub == false
string("aab") > sub == true
sub > string("aab") == false
"aab" > sub == true
sub > "aab" == false
'a' > sub == false
sub > 'a' == true

```

operator>=

Greater than or equal comparison for various objects.

```

template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
               const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>=(
    const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>=(const sub_match<BidIt>& left,
               const basic_string<typename iterator_traits<BidIt>::value_type, IOtraits, Alloc>& right);

template <class BidIt>
bool operator>=(const typename iterator_traits<BidIt>::value_type *left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator>=(const typename iterator_traits<BidIt>::value_type& left,
               const sub_match<BidIt>& right);

template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
               const typename iterator_traits<BidIt>::value_type& right);

```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `!(left < right)`.

Example


```

// std__regex__operator_ge.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub >= sub == " << std::boolalpha
        << (sub >= sub) << std::endl;

    std::cout << "string(\"aab\") >= sub == " << std::boolalpha
        << (Mystr("aab") >= sub) << std::endl;
    std::cout << "sub >= string(\"aab\") == " << std::boolalpha
        << (sub >= Mystr("aab")) << std::endl;

    std::cout << "\"aab\" >= sub == " << std::boolalpha
        << ("aab" >= sub) << std::endl;
    std::cout << "sub >= \"aab\" == " << std::boolalpha
        << (sub >= "aab") << std::endl;

    std::cout << "'a' >= sub == " << std::boolalpha
        << ('a' >= sub) << std::endl;
    std::cout << "sub >= 'a' == " << std::boolalpha
        << (sub >= 'a') << std::endl;

    return (0);
}

```

```

sub == aaa

sub >= sub == true
string("aab") >= sub == true
sub >= string("aab") == false
"aab" >= sub == true
sub >= "aab" == false
'a' >= sub == false
sub >= 'a' == true

```

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[regex_token_iterator Class](#)

[regex_traits Class](#)

[<regex> typedefs](#)

<regex> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

cmatch	cregex_iterator	cregex_token_iterator
csub_match	regex	smatch
sregex_iterator	sregex_token_iterator	ssub_match
wcmatch	wcregex_iterator	wcregex_token_iterator
wsub_match	wregex	wsmatch
wsregex_iterator	wsregex_token_iterator	wssub_match

cmatch Typedef

Type definition for char match_results.

```
typedef match_results<const char*> cmatch;
```

Remarks

The type describes a specialization of template class [match_results Class](#) for iterators of type `const char*`.

cregex_iterator Typedef

Type definition for char regex_iterator.

```
typedef regex_iterator<const char*> cregex_iterator;
```

Remarks

The type describes a specialization of template class [regex_iterator Class](#) for iterators of type `const char*`.

cregex_token_iterator Typedef

Type definition for char regex_token_iterator

```
typedef regex_token_iterator<const char*> cregex_token_iterator;
```

Remarks

The type describes a specialization of template class [regex_token_iterator Class](#) for iterators of type `const char*`.

csub_match Typedef

Type definition for char sub_match.

```
typedef sub_match<const char*> csub_match;
```

Remarks

The type describes a specialization of template class [sub_match Class](#) for iterators of type `const char*`.

regex Typedef

Type definition for char basic_regex.

```
typedef basic_regex<char> regex;
```

Remarks

The type describes a specialization of template class [basic_regex Class](#) for elements of type **char**.

NOTE

High-bit characters will have unpredictable results with `regex`. Values outside the range of 0 to 127 may result in undefined behavior.

smatch Typedef

Type definition for string match_results.

```
typedef match_results<string::const_iterator> smatch;
```

Remarks

The type describes a specialization of template class [match_results Class](#) for iterators of type `string::const_iterator`.

sregex_iterator Typedef

Type definition for string regex_iterator.

```
typedef regex_iterator<string::const_iterator> sregex_iterator;
```

Remarks

The type describes a specialization of template class [regex_iterator Class](#) for iterators of type `string::const_iterator`.

sregex_token_iterator Typedef

Type definition for string regex_token_iterator.

```
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
```

Remarks

The type describes a specialization of template class [regex_token_iterator Class](#) for iterators of type `string::const_iterator`.

ssub_match Typedef

Type definition for string sub_match.

```
typedef sub_match<string::const_iterator> ssub_match;
```

Remarks

The type describes a specialization of template class [sub_match Class](#) for iterators of type

```
string::const_iterator
```

.

wcmatch Typedef

Type definition for wchar_t match_results.

```
typedef match_results<const wchar_t*> wcmatch;
```

Remarks

The type describes a specialization of template class [match_results Class](#) for iterators of type `const wchar_t*`.

wcregex_iterator Typedef

Type definition for wchar_t regex_iterator.

```
typedef regex_iterator<const wchar_t*> wcregex_iterator;
```

Remarks

The type describes a specialization of template class [regex_iterator Class](#) for iterators of type `const wchar_t*`.

wcregex_token_iterator Typedef

Type definition for wchar_t regex_token_iterator.

```
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
```

Remarks

The type describes a specialization of template class [regex_token_iterator Class](#) for iterators of type

```
const wchar_t*
```

.

wcsub_match Typedef

Type definition for wchar_t sub_match.

```
typedef sub_match<const wchar_t*> wcsub_match;
```

Remarks

The type describes a specialization of template class [sub_match Class](#) for iterators of type `const wchar_t*`.

wregex Typedef

Type definition for `wchar_t` `basic_regex`.

```
typedef basic_regex<wchar_t> wregex;
```

Remarks

The type describes a specialization of template class [basic_regex Class](#) for elements of type **`wchar_t`**.

wsmatch Typedef

Type definition for `wstring` `match_results`.

```
typedef match_results<wstring::const_iterator> wsmatch;
```

Remarks

The type describes a specialization of template class [match_results Class](#) for iterators of type

`wstring::const_iterator`.

wsregex_iterator Typedef

Type definition for `wstring` `regex_iterator`.

```
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
```

Remarks

The type describes a specialization of template class [regex_iterator Class](#) for iterators of type

`wstring::const_iterator`.

wsregex_token_iterator Typedef

Type definition for `wstring` `regex_token_iterator`.

```
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
```

Remarks

The type describes a specialization of template class [regex_token_iterator Class](#) for iterators of type

`wstring::const_iterator`.

wssub_match Typedef

Type definition for `wstring` `sub_match`.

```
typedef sub_match<wstring::const_iterator> wssub_match;
```

Remarks

The type describes a specialization of template class [sub_match Class](#) for iterators of type

`wstring::const_iterator`.

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_token_iterator Class](#)

[regex_traits Class](#)

basic_regex Class

3/28/2019 • 6 minutes to read • [Edit Online](#)

Wraps a regular expression.

Syntax

```
template <class Elem, class RXtraits>
class basic_regex
```

Parameters

Elem

The type of elements to match.

RXtraits

Traits class for elements.

Remarks

The template class describes an object that holds a regular expression. Objects of this template class can be passed to the template functions [regex_match](#), [regex_search](#), and [regex_replace](#), along with suitable text string arguments, to search for text that matches the regular expression. There are two specializations of this template class, with the type definitions [regex](#) for elements of type **char**, and [wregex](#) for elements of type **wchar_t**.

The template argument *RXtraits* describes various important properties of the syntax of the regular expressions that the template class supports. A class that specifies these regular expression traits must have the same external interface as an object of template class [regex_traits Class](#).

Some functions take an operand sequence that defines a regular expression. You can specify such an operand sequence several ways:

`ptr` -- a null-terminated sequence (such as a C string, for *Elem* of type **char**) beginning at `ptr` (which must not be a null pointer), where the terminating element is the value `value_type()` and is not part of the operand sequence

`ptr`, `count` -- a sequence of `count` elements beginning at `ptr` (which must not be a null pointer)

`str` -- the sequence specified by the `basic_string` object `str`

`first`, `last` -- a sequence of elements delimited by the iterators `first` and `last`, in the range `[first, last)`

`right` -- the `basic_regex` object `right`

These member functions also take an argument `flags` that specifies various options for the interpretation of the regular expression in addition to those described by the *RXtraits* type.

Members

MEMBER	DEFAULT VALUE
public static const flag_type icase	regex_constants::icase

MEMBER	DEFAULT VALUE
public static const flag_type nosubs	regex_constants::nosubs
public static const flag_type optimize	regex_constants::optimize
public static const flag_type collate	regex_constants::collate
public static const flag_type ECMAScript	regex_constants::ECMAScript
public static const flag_type basic	regex_constants::basic
public static const flag_type extended	regex_constants::extended
public static const flag_type awk	regex_constants::awk
public static const flag_type grep	regex_constants::grep
public static const flag_type egrep	regex_constants::egrep
private RXtraits traits	

Constructors

CONSTRUCTOR	DESCRIPTION
basic_regex	Construct the regular expression object.

Typedefs

TYPE NAME	DESCRIPTION
flag_type	The type of syntax option flags.
locale_type	The type of the stored locale object.
value_type	The element type.

Member functions

MEMBER FUNCTION	DESCRIPTION
assign	Assigns a value to the regular expression object.
flags	Returns syntax option flags.
getloc	Returns the stored locale object.
imbue	Alters the stored locale object.
mark_count	Returns number of subexpressions matched.
swap	Swaps two regular expression objects.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns a value to the regular expression object.

Requirements

Header: <regex>

Namespace: std

Example

```
// std_regex_basic_regex.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

using namespace std;

int main()
{
    regex::value_type elem = 'x';
    regex::flag_type flag = regex::grep;

    elem = elem; // to quiet "unused" warnings
    flag = flag;

    // constructors
    regex rx0;
    cout << "match(\"abc\", \"\") == " << boolalpha
         << regex_match("abc", rx0) << endl;

    regex rx1("abcd", regex::ECMAScript);
    cout << "match(\"abc\", \"abcd\") == " << boolalpha
         << regex_match("abc", rx1) << endl;

    regex rx2("abcd", 3);
    cout << "match(\"abc\", \"abc\") == " << boolalpha
         << regex_match("abc", rx2) << endl;

    regex rx3(rx2);
    cout << "match(\"abc\", \"abc\") == " << boolalpha
         << regex_match("abc", rx3) << endl;

    string str("abcd");
    regex rx4(str);
    cout << "match(string(\"abcd\"), \"abc\") == " << boolalpha
         << regex_match("abc", rx4) << endl;

    regex rx5(str.begin(), str.end() - 1);
    cout << "match(string(\"abc\"), \"abc\") == " << boolalpha
         << regex_match("abc", rx5) << endl;
    cout << endl;

    // assignments
    rx0 = "abc";
    rx0 = rx1;
    rx0 = str;

    rx0.assign("abcd", regex::ECMAScript);
    rx0.assign("abcd", 3);
    rx0.assign(rx1);
    rx0.assign(str);
```

```

    rx0.assign(str);
    rx0.assign(str.begin(), str.end() - 1);

    rx0.swap(rx1);

    // mark_count
    cout << "\"abc\" mark_count == "
        << regex("abc").mark_count() << endl;
    cout << "\"(abc)\" mark_count == "
        << regex("(abc)").mark_count() << endl;

    // locales
    regex::locale_type loc = rx0.imbue(locale());
    cout << "getloc == imbued == " << boolalpha
        << (loc == rx0.getloc()) << endl;

    // initializer_list
    regex rx6({ 'a', 'b', 'c' }, regex::ECMAScript);
    cout << "match(\"abc\") == " << boolalpha
        << regex_match("abc", rx6);
    cout << endl;
}

```

```

match("abc", "") == false
match("abc", "abcd") == false
match("abc", "abc") == true
match("abc", "abc") == true
match(string("abcd"), "abc") == false
match(string("abc"), "abc") == true

"abc" mark_count == 0
"(abc)" mark_count == 1
getloc == imbued == true
match("abc") == true

```

basic_regex::assign

Assigns a value to the regular expression object.

```

basic_regex& assign(
    const basic_regex& right);

basic_regex& assign(
    const Elem* ptr,
    flag_type flags = ECMAScript);

basic_regex& assign(
    const Elem* ptr,
    size_type len,
    flag_type flags = ECMAScript);

basic_regex& assign(
    initializer_list<Elem> Ilist,
    flag_type flags = regex_constants::ECMAScript);

template <class STraits, class STalloc>
basic_regex& assign(
    const basic_string<Elem, STraits, STalloc>& str,
    flag_type flags = ECMAScript);

template <class InIt>
basic_regex& assign(
    InIt first, InIt last,
    flag_type flags = ECMAScript);

```

Parameters

STraits

Traits class for a string source.

STalloc

Allocator class for a string source.

InIt

Input iterator type for a range source.

right

Regex source to copy.

ptr

Pointer to beginning of sequence to copy.

flags

Syntax option flags to add while copying.

len/TD>

Length of sequence to copy.

str

String to copy.

first

Beginning of sequence to copy.

last

End of sequence to copy.

IList

The initializer_list to copy.

Remarks

The member functions each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

basic_regex::basic_regex

Construct the regular expression object.

```
basic_regex();

explicit basic_regex(
    const Elem* ptr,
    flag_type flags);

explicit basic_regex(
    const Elem* ptr,
    size_type len,
    flag_type flags);

basic_regex(
    const basic_regex& right);

basic_regex(
    initializer_list<Type> Ilist,
    flag_type flags);

template <class STraits, class STalloc>
explicit basic_regex(
    const basic_string<Elem, STraits, STalloc>& str,
    flag_type flags);

template <class InIt>
explicit basic_regex(
    InIt first,
    InIt last,
    flag_type flags);
```

Parameters

STraits

Traits class for a string source.

STalloc

Allocator class for a string source.

InIt

Input iterator type for a range source.

right

Regex source to copy.

ptr

Pointer to beginning of sequence to copy.

flags

Syntax option flags to add while copying.

len/TD>

Length of sequence to copy.

str

String to copy.

first

Beginning of sequence to copy.

last

End of sequence to copy.

/List

The initializer_list to copy.

Remarks

All constructors store a default-constructed object of type `RXtraits`.

The first constructor constructs an empty `basic_regex` object. The other constructors construct a `basic_regex` object that holds the regular expression described by the operand sequence.

An empty `basic_regex` object does not match any character sequence when passed to [regex_match](#), [regex_search](#), or [regex_replace](#).

basic_regex::flag_type

The type of syntax option flags.

```
typedef regex_constants::syntax_option_type flag_type;
```

Remarks

The type is a synonym for [regex_constants::syntax_option_type](#).

basic_regex::flags

Returns syntax option flags.

```
flag_type flags() const;
```

Remarks

The member function returns the value of the `flag_type` argument passed to the most recent call to one of the [basic_regex::assign](#) member functions or, if no such call has been made, the value passed to the constructor.

basic_regex::getloc

Returns the stored locale object.

```
locale_type getloc() const;
```

Remarks

The member function returns `traits::regex_traits::getloc()`.

basic_regex::imbue

Alters the stored locale object.

```
locale_type imbue(locale_type loc);
```

Parameters

loc

The locale object to store.

Remarks

The member function empties `*this` and returns `traits.regex_traits::imbue(loc)`.

basic_regex::locale_type

The type of the stored locale object.

```
typedef typename RXtraits::locale_type locale_type;
```

Remarks

The type is a synonym for `regex_traits::locale_type`.

basic_regex::mark_count

Returns number of subexpressions matched.

```
unsigned mark_count() const;
```

Remarks

The member function returns the number of capture groups in the regular expression.

basic_regex::operator=

Assigns a value to the regular expression object.

```
basic_regex& operator=(const basic_regex& right);

basic_regex& operator=(const Elem *str);

template <class STtraits, class STalloc>
basic_regex& operator=(const basic_string<Elem, STtraits, STalloc>& str);
```

Parameters

STtraits

Traits class for a string source.

STalloc

Allocator class for a string source.

right

Regex source to copy.

str

String to copy.

Remarks

The operators each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

basic_regex::swap

Swaps two regular expression objects.

```
void swap(basic_regex& right) throw();
```

Parameters

right

The regular expression object to swap with.

Remarks

The member function swaps the regular expressions between `*this` and *right*. It does so in constant time and throws no exceptions.

basic_regex::value_type

The element type.

```
typedef Elem value_type;
```

Remarks

The type is a synonym for the template parameter *Elem*.

See also

[<regex>](#)

[regex_match](#)

[regex_search](#)

[regex_replace](#)

[regex](#)

[wregex](#)

[regex_traits](#) Class

match_results Class

12/20/2018 • 7 minutes to read • [Edit Online](#)

Holds a sequence of submatches.

Syntax

```
template <class BidIt, class Alloc>
class match_results
```

Parameters

BidIt

The iterator type for submatches.

Alloc

The type of an allocator for managing storage.

Remarks

The template class describes an object that controls a non-modifiable sequence of elements of type `sub_match<BidIt>` generated by a regular expression search. Each element points to the subsequence that matched the capture group corresponding to that element.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>match_results</code>	Constructs the object.

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	The type of an allocator for managing storage.
<code>char_type</code>	The type of an element.
<code>const_iterator</code>	The const iterator type for submatches.
<code>const_reference</code>	The type of an element const reference.
<code>difference_type</code>	The type of an iterator difference.
<code>iterator</code>	The iterator type for submatches.
<code>reference</code>	The type of an element reference.
<code>size_type</code>	The type of a submatch count.

TYPE NAME	DESCRIPTION
<code>string_type</code>	The type of a string.
<code>value_type</code>	The type of a submatch.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Designates beginning of submatch sequence.
<code>empty</code>	Tests for no submatches.
<code>end</code>	Designates end of submatch sequence.
<code>format</code>	Formats submatches.
<code>get_allocator</code>	Returns the stored allocator.
<code>length</code>	Returns length of a submatch.
<code>max_size</code>	Gets largest number of submatches.
<code>position</code>	Get starting offset of a subgroup.
<code>prefix</code>	Gets sequence before first submatch.
<code>size</code>	Counts number of submatches.
<code>str</code>	Returns a submatch.
<code>suffix</code>	Gets sequence after last submatch.
<code>swap</code>	Swaps two <code>match_results</code> objects.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Copy a <code>match_results</code> object.
<code>operator[]</code>	Access a subobject.

Requirements

Header: `<regex>`

Namespace: `std`

Example

```
// std__regex__match_results.cpp
```

```

// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::cout << "prefix: matched == " << std::boolalpha
        << mr.prefix().matched
        << ", value == " << mr.prefix() << std::endl;
    std::cout << "whole match: " << mr.length() << " chars, value == "
        << mr.str() << std::endl;
    std::cout << "suffix: matched == " << std::boolalpha
        << mr.suffix().matched
        << ", value == " << mr.suffix() << std::endl;
    std::cout << std::endl;

    std::string fmt("\c(a*)|(b)\ matched \"%&\"\n"
        "\"(a*)\" matched \"%1\"\n"
        "\"(b)\ matched \"%2\"\n");
    std::cout << mr.format(fmt) << std::endl;
    std::cout << std::endl;

    // index through submatches
    for (size_t n = 0; n < mr.size(); ++n)
    {
        std::cout << "submatch[" << n << "]: matched == " << std::boolalpha
            << mr[n].matched <<
            " at position " << mr.position(n) << std::endl;
        std::cout << " " << mr.length(n)
            << " chars, value == " << mr[n] << std::endl;
    }
    std::cout << std::endl;

    // iterate through submatches
    for (std::cmatch::iterator it = mr.begin(); it != mr.end(); ++it)
    {
        std::cout << "next submatch: matched == " << std::boolalpha
            << it->matched << std::endl;
        std::cout << " " << it->length()
            << " chars, value == " << *it << std::endl;
    }
    std::cout << std::endl;

    // other members
    std::cout << "empty == " << std::boolalpha << mr.empty() << std::endl;

    std::cmatch::allocator_type al = mr.get_allocator();
    std::cmatch::string_type str = std::string("x");
    std::cmatch::size_type maxsiz = mr.max_size();
    std::cmatch::char_type ch = 'x';
    std::cmatch::difference_type dif = mr.begin() - mr.end();
    std::cmatch::const_iterator cit = mr.begin();
    std::cmatch::value_type val = *cit;
    std::cmatch::const_reference cref = val;
    std::cmatch::reference ref = val;

    maxsiz = maxsiz; // to quiet "unused" warnings
    if (ref == cref)
        ch = ch;
    dif = dif;

    return (0);
}

```

```

prefix: matched == true, value == x
whole match: 4 chars, value == caaa
suffix: matched == true, value == y

"c(a*)|(b)" matched "caaa"
"(a*)" matched "aaa"
"(b)" matched ""

submatch[0]: matched == true at position 1
  4 chars, value == caaa
submatch[1]: matched == true at position 2
  3 chars, value == aaa
submatch[2]: matched == false at position 6
  0 chars, value ==

next submatch: matched == true
  4 chars, value == caaa
next submatch: matched == true
  3 chars, value == aaa
next submatch: matched == false
  0 chars, value ==

empty == false

```

match_results::allocator_type

The type of an allocator for managing storage.

```
typedef Alloc allocator_type;
```

Remarks

The typedef is a synonym for the template argument *Alloc*.

match_results::begin

Designates beginning of submatch sequence.

```
const_iterator begin() const;
```

Remarks

The member function returns a random access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

match_results::char_type

The type of an element.

```
typedef typename iterator_traits<BidIt>::value_type char_type;
```

Remarks

The typedef is a synonym for the type `iterator_traits<BidIt>::value_type`, which is the element type of the character sequence that was searched.

match_results::const_iterator

The const iterator type for submatches.

```
typedef T0 const_iterator;
```

Remarks

The typedef describes an object that can serve as a constant random-access iterator for the controlled sequence.

match_results::const_reference

The type of an element const reference.

```
typedef const typename Alloc::const_reference const_reference;
```

Remarks

The typedef describes an object that can serve as a constant reference to an element of the controlled sequence.

match_results::difference_type

The type of an iterator difference.

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

Remarks

The typedef is a synonym for the type `iterator_traits<BidIt>::difference_type`; it describes an object that can represent the difference between any two iterators that point at elements of the controlled sequence.

match_results::empty

Tests for no submatches.

```
bool empty() const;
```

Remarks

The member function returns true only if the regular expression search failed.

match_results::end

Designates end of submatch sequence.

```
const_iterator end() const;
```

Remarks

The member function returns an iterator that points just beyond the end of the sequence.

match_results::format

Formats submatches.

```
template <class OutIt>
OutIt format(OutIt out,
             const string_type& fmt, match_flag_type flags = format_default) const;

string_type format(const string_type& fmt, match_flag_type flags = format_default) const;
```

Parameters

OutIt

The output iterator type.

out

The output stream to write to.

fmt

The format string.

flags

The format flags.

Remarks

Each member function generates formatted text under the control of the format *fmt*. The first member function writes the formatted text to the sequence defined by its argument *out* and returns *out*. The second member function returns a string object holding a copy of the formatted text.

To generate formatted text, literal text in the format string is ordinarily copied to the target sequence. Each escape sequence in the format string is replaced by the text that it represents. The details of the copying and replacement are controlled by the format flags passed to the function.

match_results::get_allocator

Returns the stored allocator.

```
allocator_type get_allocator() const;
```

Remarks

The member function returns a copy of the allocator object used by `*this` to allocate its `sub_match` objects.

match_results::iterator

The iterator type for submatches.

```
typedef const_iterator iterator;
```

Remarks

The type describes an object that can serve as a random-access iterator for the controlled sequence.

match_results::length

Returns length of a submatch.

```
difference_type length(size_type sub = 0) const;
```

Parameters

sub

The index of the submatch.

Remarks

The member function returns `(*this)[sub].length()`.

match_results::match_results

Constructs the object.

```
explicit match_results(const Alloc& alloc = Alloc());  
  
match_results(const match_results& right);
```

Parameters

alloc

The allocator object to store.

right

The match_results object to copy.

Remarks

The first constructor constructs a `match_results` object that holds no submatches. The second constructor constructs a `match_results` object that is a copy of *right*.

match_results::max_size

Gets largest number of submatches.

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

match_results::operator=

Copy a match_results object.

```
match_results& operator=(const match_results& right);
```

Parameters

right

The match_results object to copy.

Remarks

The member operator replaces the sequence controlled by `*this` with a copy of the sequence controlled by *right*.

match_results::operator[]

Access a subobject.

```
const_reference operator[](size_type n) const;
```

Parameters

n

Index of the submatch.

Remarks

The member function returns a reference to element *n* of the controlled sequence, or a reference to an empty

`sub_match` object if `size() <= n` or if capture group *n* was not part of the match.

match_results::position

Get starting offset of a subgroup.

```
difference_type position(size_type sub = 0) const;
```

Parameters

sub

Index of the submatch.

Remarks

The member function returns `std::distance(prefix().first, (*this)[sub].first)`, that is, the distance from the first character in the target sequence to the first character in the submatch pointed to by element `n` of the controlled sequence.

match_results::prefix

Gets sequence before first submatch.

```
const_reference prefix() const;
```

Remarks

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at the start of the target sequence and ends at `(*this)[0].first`, that is, it points to the text that precedes the matched subsequence.

match_results::reference

The type of an element reference.

```
typedef const_reference reference;
```

Remarks

The type is a synonym for the type `const_reference`.

match_results::size

Counts number of submatches.

```
size_type size() const;
```

Remarks

The member function returns one more than the number of capture groups in the regular expression that was used for the search, or zero if no search has been made.

match_results::size_type

The type of a submatch count.

```
typedef typename Alloc::size_type size_type;
```

Remarks

The type is a synonym for the type `Alloc::size_type`.

match_results::str

Returns a submatch.

```
string_type str(size_type sub = 0) const;
```

Parameters

sub

Index of the submatch.

Remarks

The member function returns `string_type((*this)[sub])`.

match_results::string_type

The type of a string.

```
typedef basic_string<char_type> string_type;
```

Remarks

The type is a synonym for the type `basic_string<char_type>`.

match_results::suffix

Gets sequence after last submatch.

```
const_reference suffix() const;
```

Remarks

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at `(*this)[size() - 1].second` and ends at the end of the target sequence, that is, it points to the text that follows the matched subsequence.

match_results::swap

Swaps two match_results objects.

```
void swap(const match_results& right) throw();
```

Parameters

right

The match_results object to swap with.

Remarks

The member function swaps the contents of `*this` and *right* in constant time and does not throw exceptions.

match_results::value_type

The type of a submatch.

```
typedef sub_match<BidIt> value_type;
```

Remarks

The typedef is a synonym for the type `sub_match<BidIt>` .

See also

[<regex>](#)

regex_constants namespace

10/31/2018 • 3 minutes to read • [Edit Online](#)

Namespace for regular expression flags.

Syntax

```
namespace regex_constants {  
    enum syntax_option_type;  
    enum match_flag_type;  
    enum error_type;  
}
```

Remarks

The namespace `regex_constants` encapsulates several flag types and their associated flag values.

<code>error_type</code>	Flags for reporting regular expression syntax errors.
<code>match_flag_type</code>	Flags for regular expression matching options.
<code>syntax_option_type</code>	Flags for selecting syntax options.

Requirements

Header: <regex>

Namespace: std

regex_constants::error_type

Flags for reporting regular expression syntax errors.

```
enum error_type  
{    // identify error  
    error_collate,  
    error_ctype,  
    error_escape,  
    error_backref,  
    error_brack,  
    error_paren,  
    error_brace,  
    error_badbrace,  
    error_range,  
    error_space,  
    error_badrepeat,  
    error_complexity,  
    error_stack,  
    error_parse,  
    error_syntax  
};
```

Remarks

The type is an enumerated type that describes an object that can hold error flags. The distinct flag values are:

`error_backref` -- the expression contained an invalid back reference

`error_badbrace` -- the expression contained an invalid count in a `{ }` expression

`error_badrepeat` -- a repeat expression (one of `'*'`, `'+'`, `'{'` in most contexts) was not preceded by an expression

`error_brace` -- the expression contained an unmatched `'{'` or `'}'`

`error_brack` -- the expression contained an unmatched `'['` or `']'`

`error_collate` -- the expression contained an invalid collating element name

`error_complexity` -- an attempted match failed because it was too complex

`error_ctype` -- the expression contained an invalid character class name

`error_escape` -- the expression contained an invalid escape sequence

`error_paren` -- the expression contained an unmatched `'('` or `')'`

`error_parse` -- the expression failed to parse

`error_range` -- the expression contained an invalid character range specifier

`error_space` -- parsing a regular expression failed because there were not enough resources available

`error_stack` -- an attempted match failed because there was not enough memory available

`error_syntax` -- parsing failed on a syntax error

`error_backref` -- the expression contained an invalid back reference

regex_constants::match_flag_type

Flags for regular expression matching options.

```
enum match_flag_type
{
    // specify matching and formatting rules
    match_default = 0x0000,
    match_not_bol = 0x0001,
    match_not_eol = 0x0002,
    match_not_bow = 0x0004,
    match_not_eow = 0x0008,
    match_any = 0x0010,
    match_not_null = 0x0020,
    match_continuous = 0x0040,
    match_prev_avail = 0x0100,
    format_default = 0x0000,
    format_sed = 0x0400,
    format_no_copy = 0x0800,
    format_first_only = 0x1000,
    _Match_not_null = 0x2000
};
```

Remarks

The type is a bitmask type that describes options to be used when matching a text sequence against a regular expression and format flags to be used when replacing text. Options can be combined with `|`.

The match options are:

match_default

`match_not_bol` -- do not treat the first position in the target sequence as the beginning of a line

`match_not_eol` -- do not treat the past-the-end position in the target sequence as the end of a line

`match_not_bow` -- do not treat the first position in the target sequence as the beginning of a word

`match_not_eow` -- do not treat the past-the-end position in the target sequence as the end of a word

`match_any` -- if more than one match is possible any match is acceptable

`match_not_null` -- do not treat an empty subsequence as a match

`match_continuous` -- do not search for matches other than at the beginning of the target sequence

`match_prev_avail` -- `--first` is a valid iterator; ignore `match_not_bol` and `match_not_bow` if set

The format flags are:

`format_default` -- use ECMAScript format rules

`format_sed` -- use sed format rules

`format_no_copy` -- do not copy text that does not match the regular expression

`format_first_only` -- do not search for matches after the first one

regex_constants::syntax_option_type

Flags for selecting syntax options.

```
enum syntax_option_type
{
    // specify RE syntax rules
    ECMAScript = 0x01,
    basic = 0x02,
    extended = 0x04,
    awk = 0x08,
    grep = 0x10,
    egrep = 0x20,
    _Gmask = 0x3F,

    icase = 0x0100,
    nosubs = 0x0200,
    optimize = 0x0400,
    collate = 0x0800
};
```

Remarks

The type is a bitmask type that describes language specifiers and syntax modifiers to be used when compiling a regular expression. Options can be combined with `|`. No more than one language specifier should be used at a time.

The language specifiers are:

`ECMAScript` -- compile as ECMAScript

`basic` -- compile as BRE

`extended` -- compile as ERE

`awk` -- compile as awk

`grep` -- compile as grep

`egrep` -- compile as egrep

The syntax modifiers are:

`icase` -- make matches case-insensitive

`nosubs` -- the implementation need not keep track of the contents of capture groups

`optimize` -- the implementation should emphasize speed of matching rather than speed of regular expression compilation

`collate` -- make matches locale-sensitive

See also

[<regex>](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_token_iterator Class](#)

[regex_traits Class](#)

[<regex> typedefs](#)

regex_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Reports a bad basic_regex object.

Syntax

```
class regex_error
: public std::runtime_error
```

Remarks

The class describes an exception object thrown to report an error in the construction or use of a `basic_regex` object.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>regex_error</code>	Constructs the object.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>code</code>	Returns the error code.

Requirements

Header: <regex>

Namespace: std

Example

```
// std_regex_regex_error.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex_error paren(std::regex_constants::error_paren);

    try
    {
        std::regex rx("(a");
    }
    catch (const std::regex_error& rerr)
    {
        std::cout << "regex error: "
            << (rerr.code() == paren.code() ? "unbalanced parentheses" : "")
            << std::endl;
    }
    catch (...)
    {
        std::cout << "unknown exception" << std::endl;
    }

    return (0);
}
```

```
regex error: unbalanced parentheses
```

regex_error::code

Returns the error code.

```
regex_constants::error_code code() const;
```

Remarks

The member function returns the value that was passed to the object's constructor.

regex_error::regex_error

Constructs the object.

```
regex_error(regex_constants::error_code error);
```

Parameters

error

The error code.

Remarks

The constructor constructs an object that holds the value *error*.

See also

[<regex>](#)

[regex_constants Class](#)

[<regex> functions](#)
[regex_iterator Class](#)
[<regex> operators](#)
[regex_token_iterator Class](#)
[regex_traits Class](#)
[<regex> typedefs](#)

regex_iterator Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Iterator class for matches.

Syntax

```
template<class BidIt,  
        class Elem = typename std::iterator_traits<BidIt>::value_type,  
        class RxTraits = regex_traits<Elem> >  
class regex_iterator
```

Parameters

BidIt

The iterator type for submatches.

Elem

The type of elements to match.

Rxtraits

Traits class for elements.

Remarks

The template class describes a constant forward iterator object. It extracts objects of type `match_results<BidIt>` by repeatedly applying its regular expression object `*pregex` to the character sequence defined by the iterator range `[begin, end)`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>regex_iterator</code>	Constructs the iterator.

Typedefs

TYPE NAME	DESCRIPTION
<code>difference_type</code>	The type of an iterator difference.
<code>iterator_category</code>	The type of the iterator category.
<code>pointer</code>	The type of a pointer to a match.
<code>reference</code>	The type of a reference to a match.
<code>regex_type</code>	The type of the regular expression to match.
<code>value_type</code>	The type of a match.

Operators

OPERATOR	DESCRIPTION
operator!=	Compares iterators for inequality.
operator*	Accesses the designated match.
operator++	Increments the iterator.
operator=	Compares iterators for equality.
operator->	Accesses the designated match.

Requirements

Header: `<regex>`

Namespace: `std`

Examples

See the following topics for examples on regular expressions:

- [regex_match](#)
- [regex_replace](#)
- [regex_search](#)
- [swap](#)

```

// std_regex_regex_iterator.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::regex_iterator<const char *> Myiter;
int main()
{
    const char *pat = "axayaz";
    Myiter::regex_type rx("a");
    Myiter next(pat, pat + strlen(pat), rx);
    Myiter end;

    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;

// other members
    Myiter it1(pat, pat + strlen(pat), rx);
    Myiter it2(it1);
    next = it1;

    Myiter::iterator_category cat = std::forward_iterator_tag();
    Myiter::difference_type dif = -3;
    Myiter::value_type mr = *it1;
    Myiter::reference ref = mr;
    Myiter::pointer ptr = &ref;

    dif = dif; // to quiet "unused" warnings
    ptr = ptr;

    return (0);
}

```

```

match == a
match == a
match == a

```

regex_iterator::difference_type

The type of an iterator difference.

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The type is a synonym for `std::ptrdiff_t`.

regex_iterator::iterator_category

The type of the iterator category.

```
typedef std::forward_iterator_tag iterator_category;
```

Remarks

The type is a synonym for `std::forward_iterator_tag`.

regex_iterator::operator!=

Compares iterators for inequality.

```
bool operator!=(const regex_iterator& right);
```

Parameters

right

The iterator to compare to.

Remarks

The member function returns `!(*this == right)`.

regex_iterator::operator*

Accesses the designated match.

```
const match_results<BidIt>& operator*();
```

Remarks

The member function returns the stored value `match`.

regex_iterator::operator++

Increments the iterator.

```
regex_iterator& operator++();  
regex_iterator& operator++(int);
```

Remarks

If the current match has no characters the first operator calls

```
regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail |  
regex_constants::match_not_null)
```

; otherwise it advances the stored value `begin` to point to the first character after the current match then calls

```
regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail)
```

. In either case, if the

search fails the operator sets the object to an end-of-sequence iterator. The operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_iterator::operator==

Compares iterators for equality.

```
bool operator==(const regex_iterator& right);
```

Parameters

right

The iterator to compare to.

Remarks

The member function returns true if `*this` and *right* are both end-of-sequence iterators or if neither is an end-of-sequence iterator and `begin == right.begin`, `end == right.end`, `pregex == right.pregex`, and `flags == right.flags`. Otherwise it returns false.

regex_iterator::operator->

Accesses the designated match.

```
const match_results<BidIt> * operator->();
```

Remarks

The member function returns the address of the stored value `match`.

regex_iterator::pointer

The type of a pointer to a match.

```
typedef match_results<BidIt> *pointer;
```

Remarks

The type is a synonym for `match_results<BidIt>*`, where `BidIt` is the template parameter.

regex_iterator::reference

The type of a reference to a match.

```
typedef match_results<BidIt>& reference;
```

Remarks

The type is a synonym for `match_results<BidIt>&`, where `BidIt` is the template parameter.

regex_iterator::regex_iterator

Constructs the iterator.

```
regex_iterator();

regex_iterator(BidIt first,
              BidIt last,
              const regex_type& re,
              regex_constants::match_flag_type f = regex_constants::match_default);
```

Parameters

first

Beginning of sequence to match.

last

End of sequence to match.

re

Regular expression for matches.

f

Flags for matches.

Remarks

The first constructor constructs an end-of-sequence iterator. The second constructor initializes the stored value

`begin` with *first*, the stored value `end` with *last*, the stored value `pregex` with `&re`, and the stored value `flags` with *f*. It then calls `regex_search(begin, end, match, *pregex, flags)`. If the search fails, the constructor sets the object to an end-of-sequence iterator.

regex_iterator::regex_type

The type of the regular expression to match.

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

Remarks

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_iterator::value_type

The type of a match.

```
typedef match_results<BidIt> value_type;
```

Remarks

The type is a synonym for `match_results<BidIt>`, where `BidIt` is the template parameter.

See also

[<regex>](#)

[regex_constants](#) Class

[regex_error](#) Class

[<regex>](#) functions

[regex_iterator](#) Class

[<regex>](#) operators

[regex_token_iterator](#) Class

[regex_traits](#) Class

[<regex>](#) typedefs

regex_token_iterator Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

Iterator class for submatches.

Syntax

```
template<class BidIt,  
        class Elem = typename std::iterator_traits<BidIt>::value_type,  
        class RxTraits = regex_traits<Elem> >  
class regex_token_iterator
```

Parameters

BidIt

The iterator type for submatches.

Elem

The type of elements to match.

RXtraits

Traits class for elements.

Remarks

The template class describes a constant forward iterator object. Conceptually, it holds a `regex_iterator` object that it uses to search for regular expression matches in a character sequence. It extracts objects of type `sub_match<BidIt>` representing the submatches identified by the index values in the stored vector `subs` for each regular expression match.

An index value of -1 designates the character sequence beginning immediately after the end of the previous regular expression match, or beginning at the start of the character sequence if there was no previous regular expression match, and extending to but not including the first character of the current regular expression match, or to the end of the character sequence if there is no current match. Any other index value `idx` designates the contents of the capture group held in `it.match[idx]`.

Members

MEMBER	DEFAULT VALUE
<code>private regex_iterator<BidIt, Elem, RXtraits> it</code>	
<code>private vector<int> subs</code>	
<code>private int pos</code>	

Constructors

CONSTRUCTOR	DESCRIPTION
<code>regex_token_iterator</code>	Constructs the iterator.

Typedefs

TYPE NAME	DESCRIPTION
<code>difference_type</code>	The type of an iterator difference.
<code>iterator_category</code>	The type of the iterator category.
<code>pointer</code>	The type of a pointer to a match.
<code>reference</code>	The type of a reference to a submatch.
<code>regex_type</code>	The type of the regular expression to match.
<code>value_type</code>	The type of a submatch.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Compares iterators for inequality.
<code>operator*</code>	Accesses the designated submatch.
<code>operator++</code>	Increments the iterator.
<code>operator==</code>	Compares iterators for equality.
<code>operator-></code>	Accesses the designated submatch.

Requirements

Header: `<regex>`

Namespace: `std`

Example


```

#include <regex>
#include <iostream>

typedef std::regex_token_iterator<const char *> Myiter;
int main()
{
    const char *pat = "aaxaayaaz";
    Myiter::regex_type rx("(a)a");
    Myiter next(pat, pat + strlen(pat), rx);
    Myiter end;

    // show whole match
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

    // show prefix before match
    next = Myiter(pat, pat + strlen(pat), rx, -1);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

    // show (a) submatch only
    next = Myiter(pat, pat + strlen(pat), rx, 1);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

    // show prefixes and submatches
    std::vector<int> vec;
    vec.push_back(-1);
    vec.push_back(1);
    next = Myiter(pat, pat + strlen(pat), rx, vec);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

    // show prefixes and whole matches
    int arr[] = {-1, 0};
    next = Myiter(pat, pat + strlen(pat), rx, arr);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

    // other members
    Myiter it1(pat, pat + strlen(pat), rx);
    Myiter it2(it1);
    next = it1;

    Myiter::iterator_category cat = std::forward_iterator_tag();
    Myiter::difference_type dif = -3;
    Myiter::value_type mr = *it1;
    Myiter::reference ref = mr;
    Myiter::pointer ptr = &ref;

    dif = dif; // to quiet "unused" warnings
    ptr = ptr;

    return (0);
}

```

```
match == aa
match == aa
match == aa

match ==
match == x
match == y
match == z

match == a
match == a
match == a

match ==
match == a
match == x
match == a
match == y
match == a
match == z

match ==
match == aa
match == x
match == aa
match == y
match == aa
match == z
```

regex_token_iterator::difference_type

The type of an iterator difference.

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The type is a synonym for `std::ptrdiff_t`.

regex_token_iterator::iterator_category

The type of the iterator category.

```
typedef std::forward_iterator_tag iterator_category;
```

Remarks

The type is a synonym for `std::forward_iterator_tag`.

regex_token_iterator::operator!=

Compares iterators for inequality.

```
bool operator!=(const regex_token_iterator& right);
```

Parameters

right

The iterator to compare to.

Remarks

The member function returns `!(*this == right)`.

regex_token_iterator::operator*

Accesses the designated submatch.

```
const sub_match<BidIt>& operator*();
```

Remarks

The member function returns a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

regex_token_iterator::operator++

Increments the iterator.

```
regex_token_iterator& operator++();  
  
regex_token_iterator& operator++(int);
```

Remarks

If the stored iterator `it` is an end-of-sequence iterator the first operator sets the stored value `pos` to the value of `subs.size()` (thus making an end-of-sequence iterator). Otherwise the operator increments the stored value `pos`; if the result is equal to the value `subs.size()` it sets the stored value `pos` to zero and increments the stored iterator `it`. If incrementing the stored iterator leaves it unequal to an end-of-sequence iterator the operator does nothing further. Otherwise, if the end of the preceding match was at the end of the character sequence the operator sets the stored value of `pos` to `subs.size()`. Otherwise, the operator repeatedly increments the stored value `pos` until `pos == subs.size()` or `subs[pos] == -1` (thus ensuring that the next dereference of the iterator will return the tail of the character sequence if one of the index values is -1). In all cases the operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_token_iterator::operator==

Compares iterators for equality.

```
bool operator==(const regex_token_iterator& right);
```

Parameters

right

The iterator to compare to.

Remarks

The member function returns `it == right.it && subs == right.subs && pos == right.pos`.

regex_token_iterator::operator->

Accesses the designated submatch.

```
const sub_match<BidIt> * operator->();
```

Remarks

The member function returns a pointer to a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

regex_token_iterator::pointer

The type of a pointer to a match.

```
typedef sub_match<BidIt> *pointer;
```

Remarks

The type is a synonym for `sub_match<BidIt>*`, where `BidIt` is the template parameter.

regex_token_iterator::reference

The type of a reference to a submatch.

```
typedef sub_match<BidIt>& reference;
```

Remarks

The type is a synonym for `sub_match<BidIt>&`, where `BidIt` is the template parameter.

regex_token_iterator::regex_token_iterator

Constructs the iterator.

```
regex_token_iterator();

regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, int submatch = 0,
    regex_constants::match_flag_type f = regex_constants::match_default);

regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const vector<int> submatches,
    regex_constants::match_flag_type f = regex_constants::match_default);

template <std::size_t N>
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const int (&submatches)[N],
    regex_constants::match_flag_type f = regex_constants::match_default);
```

Parameters

first

Beginning of sequence to match.

last

End of sequence to match.

re

Regular expression for matches.

f

Flags for matches.

Remarks

The first constructor constructs an end-of-sequence iterator.

The second constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds exactly one integer, with value `submatch`, and whose stored value `pos` is zero. Note: the resulting object extracts the submatch identified by the index value `submatch` for each successful regular expression match.

The third constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds a copy of the constructor argument `submatches`, and whose stored value `pos` is zero.

The fourth constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds the `N` values pointed to by the constructor argument `submatches`, and whose stored value `pos` is zero.

regex_token_iterator::regex_type

The type of the regular expression to match.

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

Remarks

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_token_iterator::value_type

The type of a submatch.

```
typedef sub_match<BidIt> value_type;
```

Remarks

The type is a synonym for `sub_match<BidIt>`, where `BidIt` is the template parameter.

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_traits Class](#)

[<regex> typedefs](#)

regex_traits Class

10/31/2018 • 7 minutes to read • [Edit Online](#)

Describes characteristics of elements for matching.

Syntax

```
template<class Elem>
class regex_traits
```

Parameters

Elem

The character element type to describe.

Remarks

The template class describes various regular expression traits for type *Elem*. The template class [basic_regex Class](#) uses this information to manipulate elements of type *Elem*.

Each `regex_traits` object holds an object of type `regex_traits::locale` which is used by some of its member functions. The default locale is a copy of `regex_traits::locale()`. The member function `imbue` replaces the locale object, and the member function `getloc` returns a copy of the locale object.

Constructors

CONSTRUCTOR	DESCRIPTION
regex_traits	Constructs the object.

Typedefs

TYPE NAME	DESCRIPTION
char_class_type	The type of character class designators.
char_type	The type of an element.
locale_type	The type of the stored locale object.
size_type	The type of a sequence length.
string_type	The type of a string of elements.

Member functions

MEMBER FUNCTION	DESCRIPTION
getloc	Returns the stored locale object.

MEMBER FUNCTION	DESCRIPTION
<code>imbue</code>	Alters the stored locale object.
<code>istype</code>	Tests for class membership.
<code>length</code>	Returns the length of a null-terminated sequence.
<code>lookup_classname</code>	Maps a sequence to a character class.
<code>lookup_collatename</code>	Maps a sequence to a collating element.
<code>transform</code>	Converts to equivalent ordered sequence.
<code>transform_primary</code>	Converts to equivalent caseless ordered sequence.
<code>translate</code>	Converts to equivalent matching element.
<code>translate_nocase</code>	Converts to equivalent caseless matching element.
<code>value</code>	Converts an element to a digit value.

Requirements

Header: <regex>

Namespace: std

Example

```

// std_regex_traits.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::regex_traits<char> Mytr;
int main()
{
    Mytr tr;

    Mytr::char_type ch = tr.translate('a');
    std::cout << "translate('a') == 'a' == " << std::boolalpha
        << (ch == 'a') << std::endl;

    std::cout << "nocase 'a' == 'A' == " << std::boolalpha
        << (tr.translate_nocase('a') == tr.translate_nocase('A'))
        << std::endl;

    const char *lbegin = "abc";
    const char *lend = lbegin + strlen(lbegin);
    Mytr::size_type size = tr.length(lbegin);
    std::cout << "length(\"abc\") == " << size << std::endl;

    Mytr::string_type str = tr.transform(lbegin, lend);
    std::cout << "transform(\"abc\") < \"abc\" == " << std::boolalpha
        << (str < "abc") << std::endl;

    const char *ubegin = "ABC";
    const char *uend = ubegin + strlen(ubegin);
    std::cout << "primary \"ABC\" < \"abc\" == " << std::boolalpha
        << (tr.transform_primary(ubegin, uend) <
            tr.transform_primary(lbegin, lend))
        << std::endl;

    const char *dig = "digit";
    Mytr::char_class_type cl = tr.lookup_classname(dig, dig + 5);
    std::cout << "class digit == d == " << std::boolalpha
        << (cl == tr.lookup_classname(dig, dig + 1))
        << std::endl;

    std::cout << "'3' is digit == " << std::boolalpha
        << tr.istype('3', tr.lookup_classname(dig, dig + 5))
        << std::endl;

    std::cout << "hex C == " << tr.value('C', 16) << std::endl;

    // other members
    str = tr.lookup_collatename(dig, dig + 5);

    Mytr::locale_type loc = tr.getloc();
    tr.imbue(loc);

    return (0);
}

```

```

translate('a') == 'a' == true
nocase 'a' == 'A' == true
length("abc") == 3
transform("abc") < "abc" == false
primary "ABC" < "abc" == false
class digit == d == true
'3' is digit == true
hex C == 12

```


regex_traits::char_class_type

The type of character class designators.

```
typedef T8 char_class_type;
```

Remarks

The type is a synonym for an unspecified type that designates character classes. Values of this type can be combined using the `|` operator to designate character classes that are the union of the classes designated by the operands.

regex_traits::char_type

The type of an element.

```
typedef Elem char_type;
```

Remarks

The typedef is a synonym for the template argument `Elem`.

regex_traits::getloc

Returns the stored locale object.

```
locale_type getloc() const;
```

Remarks

The member function returns the stored `locale` object.

regex_traits::imbue

Alters the stored locale object.

```
locale_type imbue(locale_type loc);
```

Parameters

loc

The locale object to store.

Remarks

The member function copies *loc* to the stored `locale` object and returns a copy of the previous value of the stored `locale` object.

regex_traits::isctype

Tests for class membership.

```
bool isctype(char_type ch, char_class_type cls) const;
```

Parameters

ch

The element to test.

cls

The classes to test for.

Remarks

The member function returns true only if the character *ch* is in the character class designated by *cls*.

regex_traits::length

Returns the length of a null-terminated sequence.

```
static size_type length(const char_type *str);
```

Parameters

str

The null-terminated sequence.

Remarks

The static member function returns `std::char_traits<char_type>::length(str)`.

regex_traits::locale_type

The type of the stored locale object.

```
typedef T7 locale_type;
```

Remarks

The typedef is a synonym for a type that encapsulates locales. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::locale`.

regex_traits::lookup_classname

Maps a sequence to a character class.

```
template <class FwdIt>
char_class_type lookup_classname(FwdIt first, FwdIt last) const;
```

Parameters

first

Beginning of sequence to look up.

last

End of sequence to look up.

Remarks

The member function returns a value that designates the character class named by the character sequence pointed to by its arguments. The value does not depend on the case of the characters in the sequence.

The specialization `regex_traits<char>` recognizes the names `"d"`, `"s"`, `"w"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"`, all without regard

to case.

The specialization `regex_traits<wchar_t>` recognizes the names `L"d"`, `L"s"`, `L"w"`, `L"alnum"`, `L"alpha"`, `L"blank"`, `L"cntrl"`, `L"digit"`, `L"graph"`, `L"lower"`, `L"print"`, `L"punct"`, `L"space"`, `L"upper"`, and `L"xdigit"`, all without regard to case.

regex_traits::lookup_collatename

Maps a sequence to a collating element.

```
template <class FwdIt>
string_type lookup_collatename(FwdIt first, FwdIt last) const;
```

Parameters

first

Beginning of sequence to look up.

last

End of sequence to look up.

Remarks

The member function returns a string object containing the collating element corresponding to the sequence `[first, last)`, or an empty string if the sequence is not a valid collating element.

regex_traits::regex_traits

Constructs the object.

```
regex_traits();
```

Remarks

The constructor constructs an object whose stored `locale` object is initialized to the default locale.

regex_traits::size_type

The type of a sequence length.

```
typedef T6 size_type;
```

Remarks

The typedef is a synonym for an unsigned integral type. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::size_t`.

The typedef is a synonym for `std::size_t`.

regex_traits::string_type

The type of a string of elements.

```
typedef basic_string<Elem> string_type;
```

Remarks

The typedef is a synonym for `basic_string<Elem>`.

regex_traits::transform

Converts to equivalent ordered sequence.

```
template <class FwdIt>
string_type transform(FwdIt first, FwdIt last) const;
```

Parameters

first

Beginning of sequence to transform.

last

End of sequence to transform.

Remarks

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform(first1, last1) < transform(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)`.

regex_traits::transform_primary

Converts to equivalent caseless ordered sequence.

```
template <class FwdIt>
string_type transform_primary(FwdIt first, FwdIt last) const;
```

Parameters

first

Beginning of sequence to transform.

last

End of sequence to transform.

Remarks

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform_primary(first1, last1) < transform_primary(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)` without regard for case or accents.

regex_traits::translate

Converts to equivalent matching element.

```
char_type translate(char_type ch) const;
```

Parameters

ch

The element to convert.

Remarks

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate(ch1) == translate(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a locale-sensitive match.

regex_traits::translate_nocase

Converts to equivalent caseless matching element.

```
char_type translate_nocase(char_type ch) const;
```

Parameters

ch

The element to convert.

Remarks

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate_nocase(ch1) == translate_nocase(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a case-insensitive match.

regex_traits::value

Converts an element to a digit value.

```
int value(Elem ch, int radix) const;
```

Parameters

ch

The element to convert.

radix

The arithmetic base to use.

Remarks

The member function returns the value represented by the character *ch* in the base *radix*, or -1 if *ch* is not a valid digit in the base *radix*. The function will only be called with a *radix* argument of 8, 10, or 16.

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_token_iterator Class](#)

[<regex> typedefs](#)

[regex_traits<char> Class](#)

[regex_traits<wchar_t> Class](#)

regex_traits<char> Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Specialization of `regex_traits` for **char**.

Syntax

```
template <>
class regex_traits<char>
```

Remarks

The class is an explicit specialization of template class [regex_traits](#) for elements of type **char** (so that it can take advantage of library functions that manipulate objects of this type).

Requirements

Header: <regex>

Namespace: std

See also

- [<regex>](#)
- [regex_constants Class](#)
- [regex_error Class](#)
- [<regex> functions](#)
- [regex_iterator Class](#)
- [<regex> operators](#)
- [regex_token_iterator Class](#)
- [regex_traits Class](#)
- [<regex> typedefs](#)

regex_traits<wchar_t> Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Specialization of `regex_traits` for **wchar_t**.

Syntax

```
template <>
class regex_traits<wchar_t>
```

Remarks

The class is an explicit specialization of template class [regex_traits](#) for elements of type **wchar_t** (so that it can take advantage of library functions that manipulate objects of this type).

Requirements

Header: <regex>

Namespace: std

See also

[<regex>](#)

[regex_constants Class](#)

[regex_error Class](#)

[<regex> functions](#)

[regex_iterator Class](#)

[<regex> operators](#)

[regex_token_iterator Class](#)

[regex_traits Class](#)

[<regex> typedefs](#)

sub_match Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes a submatch.

Syntax

```
template <class BidIt>
class sub_match
: public std::pair<BidIt, BidIt>
```

Parameters

BidIt

The iterator type for submatches.

Remarks

The template class describes an object that designates a sequence of characters that matched a capture group in a call to [regex_match](#) or to [regex_search](#). Objects of type [match_results Class](#) hold an array of these objects, one for each capture group in the regular expression that was used in the search.

If the capture group was not matched the object's data member `matched` holds false, and the two iterators `first` and `second` (inherited from the base `std::pair`) are equal. If the capture group was matched, `matched` holds true, the iterator `first` points to the first character in the target sequence that matched the capture group, and the iterator `second` points one position past the last character in the target sequence that matched the capture group. Note that for a zero-length match the member `matched` holds true, the two iterators will be equal, and both will point to the position of the match.

A zero-length match can occur when a capture group consists solely of an assertion, or of a repetition that allows zero repeats. For example:

"^" matches the target sequence "a"; the `sub_match` object corresponding to capture group 0 holds iterators that both point to the first character in the sequence.

"b(a*)b" matches the target sequence "bb"; the `sub_match` object corresponding to capture group 1 holds iterators that both point to the second character in the sequence.

Typedefs

TYPE NAME	DESCRIPTION
difference_type	The type of an iterator difference.
iterator	The type of an iterator.
value_type	The type of an element.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>compare</code>	Compare submatch against a sequence.
<code>length</code>	Returns the length of a submatch.
<code>matched</code>	Indicates if match succeeded.
<code>str</code>	Converts submatch to a string.

Operators

OPERATOR	DESCRIPTION
<code>operator basic_string<value_type></code>	Casts submatch to a string.

Example

```
// std__regex__sub_match.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "matched == " << std::boolalpha
              << sub.matched << std::endl;
    std::cout << "length == " << sub.length() << std::endl;

    std::csub_match::difference_type dif = std::distance(sub.first, sub.second);
    std::cout << "difference == " << dif << std::endl;

    std::csub_match::iterator first = sub.first;
    std::csub_match::iterator last = sub.second;
    std::cout << "range == " << std::string(first, last)
              << std::endl;
    std::cout << "string == " << sub << std::endl;

    std::csub_match::value_type const *ptr = "aab";
    std::cout << "compare(\"aab\") == "
              << sub.compare(ptr) << std::endl;
    std::cout << "compare(string) == "
              << sub.compare(std::string("AAA")) << std::endl;
    std::cout << "compare(sub) == "
              << sub.compare(sub) << std::endl;

    return (0);
}
```

```
matched == true
length == 3
difference == 3
range == aaa
string == aaa
compare("aab") == -1
compare(string) == 1
compare(sub) == 0
```

Requirements

Header: <regex>

Namespace: std

sub_match::compare

Compare submatch against a sequence.

```
int compare(const sub_match& right) const;
int compare(const basic_string<value_type>& str) const;
int compare(const value_type *ptr) const;
```

Parameters

right

The submatch to compare to.

str

The string to compare to.

ptr

The null-terminated sequence to compare to.

Remarks

The first member function compares the matched sequence `[first, second)` to the matched sequence `[right.first, right.second)`. The second member function compares the matched sequence `[first, second)` to the character sequence `[right.begin(), right.end())`. The third member function compares the matched sequence `[first, second)` to the character sequence `[right, right + std::char_traits<value_type>::length(right))`.

Each function returns:

a negative value if the first differing value in the matched sequence compares less than the corresponding element in the operand sequence (as determined by `std::char_traits<value_type>::compare`), or if the two have a common prefix but the target sequence is longer

zero if the two compare equal element by element and have the same length

a positive value otherwise

sub_match::difference_type

The type of an iterator difference.

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

Remarks

The typedef is a synonym for `iterator_traits<BidIt>::difference_type` .

sub_match::iterator

The type of an iterator.

```
typedef BidIt iterator;
```

Remarks

The typedef is a synonym for the template type argument `Bidit` .

sub_match::length

Returns the length of a submatch.

```
difference_type length() const;
```

Remarks

The member function returns the length of the matched sequence, or zero if there was no matched sequence.

sub_match::matched

Indicates if match succeeded.

```
bool matched;
```

Remarks

The member holds **true** only if the capture group associated with `*this` was part of the regular expression match.

sub_match::operator basic_string<value_type>

Casts submatch to a string.

```
operator basic_string<value_type>() const;
```

Remarks

The member operator returns `str()` .

sub_match::str

Converts submatch to a string.

```
basic_string<value_type> str() const;
```

Remarks

The member function returns `basic_string<value_type>(first, second)` .

sub_match::value_type

The type of an element.

```
typedef typename iterator_traits<BidIt>::value_type value_type;
```

Remarks

The typedef is a synonym for `iterator_traits<BidIt>::value_type` .

See also

[<regex>](#)

[sub_match](#)

<scoped_allocator>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class `scoped_allocator`.

Syntax

```
#include <scoped_allocator>
```

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the <code>scoped_allocator</code> object on the left side of the operator is not equal to the list object on the right side.
<code>operator==</code>	Tests if the <code>scoped_allocator</code> object on the left side of the operator is equal to the list object on the right side.

Classes

CLASS	DESCRIPTION
<code>scoped_allocator_adaptor</code> Class	A template class that encapsulates a nest of one or more allocators.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<scoped_allocator> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

`operator!=`

`operator==`

operator!=

Tests two `scoped_allocator_adaptor` objects for inequality.

```
template <class Outer, class... Inner>
bool operator!=(
    const scoped_allocator_adaptor<Outer, Inner...>& left,
    const scoped_allocator_adaptor<Outer, Inner...>& right) noexcept;
```

Parameters

left

The left `scoped_allocator_adaptor` object.

right

The right `scoped_allocator_adaptor` object.

Return Value

`!(left == right)`

operator==

Tests two `scoped_allocator_adaptor` objects for equality.

```
template <class Outer, class... Inner>
bool operator==(
    const scoped_allocator_adaptor<Outer, Inner...>& left,
    const scoped_allocator_adaptor<Outer, Inner...>& right) noexcept;
```

Parameters

left

The left `scoped_allocator_adaptor` object.

right

The right `scoped_allocator_adaptor` object.

Return Value

`left.outer_allocator() == right.outer_allocator() && left.inner_allocator() == right.inner_allocator()`

See also

[<scoped_allocator>](#)

scoped_allocator_adaptor Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

Represents a nest of allocators.

Syntax

```
template <class Outer, class... Inner>
class scoped_allocator_adaptor;
```

Remarks

The template class encapsulates a nest of one or more allocators. Each such class has an outermost allocator of type `outer_allocator_type`, a synonym for `Outer`, which is a public base of the `scoped_allocator_adaptor` object. `Outer` is used to allocate memory to be used by a container. You can obtain a reference to this allocator base object by calling `outer_allocator`.

The remainder of the nest has type `inner_allocator_type`. An inner allocator is used to allocate memory for elements within a container. You can obtain a reference to the stored object of this type by calling `inner_allocator`. If `Inner...` is not empty, `inner_allocator_type` has type `scoped_allocator_adaptor<Inner...>`, and `inner_allocator` designates a member object. Otherwise, `inner_allocator_type` has type `scoped_allocator_adaptor<Outer>`, and `inner_allocator` designates the entire object.

The nest behaves as if it has arbitrary depth, replicating its innermost encapsulated allocator as needed.

Several concepts that are not a part of the visible interface aid in describing the behavior of this template class. An *outermost allocator* mediates all calls to the construct and destroy methods. It is effectively defined by the recursive function `OUTERMOST(X)`, where `OUTERMOST(X)` is one of the following.

- If `X.outer_allocator()` is well formed, then `OUTERMOST(X)` is `OUTERMOST(X.outer_allocator())`.
- Otherwise, `OUTERMOST(X)` is `X`.

Three types are defined for the sake of exposition:

TYPE	DESCRIPTION
<code>Outermost</code>	The type of <code>OUTERMOST(*this)</code> .
<code>Outermost_traits</code>	<code>allocator_traits<Outermost></code>
<code>Outer_traits</code>	<code>allocator_traits<Outer></code>

Constructors

NAME	DESCRIPTION
<code>scoped_allocator_adaptor</code>	Constructs a <code>scoped_allocator_adaptor</code> object.

Typedefs

NAME	DESCRIPTION
<code>const_pointer</code>	This type is a synonym for the <code>const_pointer</code> that is associated with the allocator <code>Outer</code> .
<code>const_void_pointer</code>	This type is a synonym for the <code>const_void_pointer</code> that is associated with the allocator <code>Outer</code> .
<code>difference_type</code>	This type is a synonym for the <code>difference_type</code> that is associated with the allocator <code>Outer</code> .
<code>inner_allocator_type</code>	This type is a synonym for the type of the nested adaptor <code>scoped_allocator_adaptor<Inner...></code> .
<code>outer_allocator_type</code>	This type is a synonym for the type of the base allocator <code>Outer</code> .
<code>pointer</code>	This type is a synonym for the <code>pointer</code> associated with the allocator <code>Outer</code> .
<code>propagate_on_container_copy_assignment</code>	The type holds true only if <code>Outer_traits::propagate_on_container_copy_assignment</code> holds true or <code>inner_allocator_type::propagate_on_container_copy_assignment</code> holds true.
<code>propagate_on_container_move_assignment</code>	The type holds true only if <code>Outer_traits::propagate_on_container_move_assignment</code> holds true or <code>inner_allocator_type::propagate_on_container_move_assignment</code> holds true.
<code>propagate_on_container_swap</code>	The type holds true only if <code>Outer_traits::propagate_on_container_swap</code> holds true or <code>inner_allocator_type::propagate_on_container_swap</code> holds true.
<code>size_type</code>	This type is a synonym for the <code>size_type</code> associated with the allocator <code>Outer</code> .
<code>value_type</code>	This type is a synonym for the <code>value_type</code> associated with the allocator <code>Outer</code> .
<code>void_pointer</code>	This type is a synonym for the <code>void_pointer</code> associated with the allocator <code>Outer</code> .

Structs

NAME	DESCRIPTION
scoped_allocator_adaptor::rebind Struct	Defines the type <code>Outer::rebind<Other>::other</code> as a synonym for <code>scoped_allocator_adaptor<Other, Inner...></code> .

Methods

NAME	DESCRIPTION
<code>allocate</code>	Allocates memory by using the <code>Outer</code> allocator.
<code>construct</code>	Constructs an object.
<code>deallocate</code>	Deallocates objects by using the outer allocator.
<code>destroy</code>	Destroys a specified object.
<code>inner_allocator</code>	Retrieves a reference to the stored object of type <code>inner_allocator_type</code> .
<code>max_size</code>	Determines the maximum number of objects that can be allocated by the outer allocator.
<code>outer_allocator</code>	Retrieves a reference to the stored object of type <code>outer_allocator_type</code> .
<code>select_on_container_copy_construction</code>	Creates a new <code>scoped_allocator_adaptor</code> object with each stored allocator object initialized by calling <code>select_on_container_copy_construction</code> for each corresponding allocator.

Requirements

Header: `<scoped_allocator>`

Namespace: `std`

`scoped_allocator_adaptor::allocate`

Allocates memory by using the `Outer` allocator.

```
pointer allocate(size_type count);
pointer allocate(size_type count, const_void_pointer hint);
```

Parameters

count

The number of elements for which sufficient storage is to be allocated.

hint

A pointer that might assist the allocator object by locating the address of an object allocated prior to the request.

Return Value

The first member function returns `Outer_traits::allocate(outer_allocator(), count)`. The second member function returns `Outer_traits::allocate(outer_allocator(), count, hint)`.

`scoped_allocator_adaptor::construct`

Constructs an object.

```

template <class Ty, class... Atypes>
void construct(Ty* ptr, Atypes&&... args);

template <class Ty1, class Ty2, class... Atypes1, class... Atypes2>
void construct(pair<Ty1, Ty2>* ptr, piecewise_construct_t,
    tuple<Atypes1&&...>
first, tuple<Atypes1&&...> second);

template <class Ty1, class Ty2>
void construct(pair<Ty1, Ty2>* ptr);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr,
    class Uy1&& first, class Uy2&& second);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr, const pair<Uy1, Uy2>& right);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr, pair<Uy1, Uy2>&& right);

```

Parameters

ptr

A pointer to the memory location where the object is to be constructed.

args

A list of arguments.

first

An object of the first type in a pair.

second

An object of the second type in a pair.

right

An existing object to be moved or copied.

Remarks

The first method constructs the object at *ptr* by calling

`Outermost_traits::construct(OUTERMOST(*this), ptr, xargs...)`, where `xargs...` is one of the following.

- If `uses_allocator<Ty, inner_allocator_type>` holds false, then `xargs...` is `args...`.
- If `uses_allocator<Ty, inner_allocator_type>` holds true, and `is_constructible<Ty, allocator_arg_t, inner_allocator_type, args...>` holds true, then `xargs...` is `allocator_arg, inner_allocator(), args...`.
- If `uses_allocator<Ty, inner_allocator_type>` holds true, and `is_constructible<Ty, args..., inner_allocator()>` holds true, then `xargs...` is `args..., inner_allocator()`.

The second method constructs the pair object at *ptr* by calling

`Outermost_traits::construct(OUTERMOST(*this), &ptr->first, xargs...)`, where `xargs...` is `first...` modified as in the above list, and `Outermost_traits::construct(OUTERMOST(*this), &ptr->second, xargs...)`, where `xargs...` is `second...` modified as in the above list.

The third method behaves the same as `this->construct(ptr, piecewise_construct, tuple<>, tuple<>)`.

The fourth method behaves the same as

```
this->construct(ptr, piecewise_construct, forward_as_tuple(std::forward<Uy1>(first),
forward_as_tuple(std::forward<Uy2>(second)))
```

The fifth method behaves the same as

```
this->construct(ptr, piecewise_construct, forward_as_tuple(right.first), forward_as_tuple(right.second)).
```

The sixth method behaves the same as

```
this->construct(ptr, piecewise_construct, forward_as_tuple(std::forward<Uy1>(right.first),
forward_as_tuple(std::forward<Uy2>(right.second)))
```

scoped_allocator_adaptor::deallocate

Deallocates objects by using the outer allocator.

```
void deallocate(pointer ptr, size_type count);
```

Parameters

ptr

A pointer to the starting location of the objects to be deallocated.

count

The number of objects to deallocate.

scoped_allocator_adaptor::destroy

Destroys a specified object.

```
template <class Ty>
void destroy(Ty* ptr)
```

Parameters

ptr

A pointer to the object to be destroyed.

Return Value

```
Outermost_traits::destroy(OUTERMOST(*this), ptr)
```

scoped_allocator_adaptor::inner_allocator

Retrieves a reference to the stored object of type `inner_allocator_type`.

```
inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
```

Return Value

A reference to the stored object of type `inner_allocator_type`.

scoped_allocator_adaptor::max_size

Determines the maximum number of objects that can be allocated by the outer allocator.

```
size_type max_size();
```

Return Value

```
Outer_traits::max_size(outer_allocator())
```

scoped_allocator_adaptor::outer_allocator

Retrieves a reference to the stored object of type `outer_allocator_type`.

```
outer_allocator_type& outer_allocator() noexcept;  
const outer_allocator_type& outer_allocator() const noexcept;
```

Return Value

A reference to the stored object of type `outer_allocator_type`.

scoped_allocator_adaptor::rebind Struct

Defines the type `Outer::rebind<Other>::other` as a synonym for `scoped_allocator_adaptor<Other, Inner...>`.

```
struct rebind{ typedef Other_traits::rebind<Other> Other_alloc; typedef scoped_allocator_adaptor<Other_alloc,  
Inner...> other; };
```

scoped_allocator_adaptor::scoped_allocator_adaptor Constructor

Constructs a `scoped_allocator_adaptor` object.

```
scoped_allocator_adaptor();  
  
scoped_allocator_adaptor(const scoped_allocator_adaptor& right) noexcept;  
template <class Outer2>  
scoped_allocator_adaptor(  
const scoped_allocator_adaptor<Outer2, Inner...>& right) noexcept;  
template <class Outer2>  
scoped_allocator_adaptor(  
scoped_allocator_adaptor<Outer2, Inner...>&& right) noexcept;  
template <class Outer2>  
scoped_allocator_adaptor(Outer2&& al,  
const Inner&... rest) noexcept;
```

Parameters

right

An existing `scoped_allocator_adaptor`.

al

An existing allocator to be used as the outer allocator.

rest

A list of allocators to be used as the inner allocators.

Remarks

The first constructor default constructs its stored allocator objects. Each of the next three constructors constructs its stored allocator objects from the corresponding objects in *right*. The last constructor constructs its stored allocator objects from the corresponding arguments in the argument list.

scoped_allocator_adaptor::select_on_container_copy_construction

Creates a new `scoped_allocator_adaptor` object with each stored allocator object initialized by calling `select_on_container_copy_construction` for each corresponding allocator.

```
scoped_allocator_adaptor select_on_container_copy_construction();
```

Return Value

This method effectively returns

```
scoped_allocator_adaptor(Outer_traits::select_on_container_copy_construction(*this),  
inner_allocator().select_on_container_copy_construction())
```

. The result is a new `scoped_allocator_adaptor` object with each stored allocator object initialized by calling `al.select_on_container_copy_construction()` for the corresponding allocator *al*.

See also

[Header Files Reference](#)

<set>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template classes set and multiset and their supporting templates.

Syntax

```
#include <set>
```

Members

Operators

SET VERSION	MULTISET VERSION	DESCRIPTION
<code>operator!= (set)</code>	<code>operator!= (multiset)</code>	Tests if the set or multiset object on the left side of the operator is not equal to the set or multiset object on the right side.
<code>operator< (set)</code>	<code>operator< (multiset)</code>	Tests if the set or multiset object on the left side of the operator is less than the set or multiset object on the right side.
<code>operator<= (set)</code>	<code>operator<= (multiset)</code>	Tests if the set or multiset object on the left side of the operator is less than or equal to the set or multiset object on the right side.
<code>operator== (set)</code>	<code>operator== (multiset)</code>	Tests if the set or multiset object on the left side of the operator is equal to the set or multiset object on the right side.
<code>operator> (set)</code>	<code>operator> (multiset)</code>	Tests if the set or multiset object on the left side of the operator is greater than the set or multiset object on the right side.
<code>operator>= (set)</code>	<code>operator>= (multiset)</code>	Tests if the set or multiset object on the left side of the operator is greater than or equal to the set or multiset object on the right side.

Specialized Template Functions

SET VERSION	MULTISET VERSION	DESCRIPTION
<code>swap</code>	<code>swap (multiset)</code>	Exchanges the elements of two sets or multisets.

Classes

CLASS	DESCRIPTION
set Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered.
multiset Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<set> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

[swap \(map\)](#)

[swap \(multiset\)](#)

swap (map)

Exchanges the elements of two sets.

```
template <class Key, class Traits, class Allocator>
void swap(set<Key, Traits, Allocator>& left, set<Key, Traits, Allocator>& right);
```

Parameters

right

The set providing the elements to be swapped, or the set whose elements are to be exchanged with those of the set *left*.

left

The set whose elements are to be exchanged with those of the set *right*.

Remarks

The template function is an algorithm specialized on the container class `set` to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

```
template < classT> void swap( T&, T&)
```

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class `set::swap` for an example of the use of the template version of `swap`.

swap (multiset)

Exchanges the elements of two multisets.

```
template <class Key, class Traits, class Allocator>
void swap(multiset<Key, Traits, Allocator>& left, multiset<Key, Traits, Allocator>& right);
```

Parameters

right

The multiset providing the elements to be swapped, or the multiset whose elements are to be exchanged with those of the multiset *left*.

left

The multiset whose elements are to be exchanged with those of the multiset *right*.

Remarks

The template function is an algorithm specialized on the container class multiset to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

```
template < classT> void swap( T&, T&)
```

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class `multiset::swap` for an example of the use of the template version of

```
swap .
```

See also

[<set>](#)

<set> operators

3/28/2019 • 15 minutes to read • [Edit Online](#)

<code>operator!= (set)</code>	<code>operator> (set)</code>	<code>operator>= (set)</code>
<code>operator< (set)</code>	<code>operator<= (set)</code>	<code>operator== (set)</code>
<code>operator!= (multiset)</code>	<code>operator> (multiset)</code>	<code>operator>= (multiset)</code>
<code>operator< (multiset)</code>	<code>operator<= (multiset)</code>	<code>operator== (multiset)</code>

operator!= (set)

Tests if the set object on the left side of the operator is not equal to the set object on the right side.

```
bool operator!=(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the sets are not equal; **false** if sets are equal.

Remarks

The comparison between set objects is based on a pairwise comparison between their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// set_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 != s2 )
        cout << "The sets s1 and s2 are not equal." << endl;
    else
        cout << "The sets s1 and s2 are equal." << endl;

    if ( s1 != s3 )
        cout << "The sets s1 and s3 are not equal." << endl;
    else
        cout << "The sets s1 and s3 are equal." << endl;
}
/* Output:
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
*/

```

operator< (set)

Tests if the set object on the left side of the operator is less than the set object on the right side.

```
bool operator<(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the set on the left side of the operator is strictly less than the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// set_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
    }

    if ( s1 < s2 )
        cout << "The set s1 is less than the set s2." << endl;
    else
        cout << "The set s1 is not less than the set s2." << endl;

    if ( s1 < s3 )
        cout << "The set s1 is less than the set s3." << endl;
    else
        cout << "The set s1 is not less than the set s3." << endl;
}
/* Output:
The set s1 is less than the set s2.
The set s1 is not less than the set s3.
*/

```

operator<= (set)

Tests if the set object on the left side of the operator is less than or equal to the set object on the right side.

```
bool operator!<=(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the set on the left side of the operator is less than or equal to the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// set_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3, s4;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
        s4.insert ( i );
    }

    if ( s1 <= s2 )
        cout << "Set s1 is less than or equal to the set s2." << endl;
    else
        cout << "The set s1 is greater than the set s2." << endl;

    if ( s1 <= s3 )
        cout << "Set s1 is less than or equal to the set s3." << endl;
    else
        cout << "The set s1 is greater than the set s3." << endl;

    if ( s1 <= s4 )
        cout << "Set s1 is less than or equal to the set s4." << endl;
    else
        cout << "The set s1 is greater than the set s4." << endl;
}
/* Output:
Set s1 is less than or equal to the set s2.
The set s1 is greater than the set s3.
Set s1 is less than or equal to the set s4.
*/

```

operator== (set)

Tests if the set object on the left side of the operator is equal to the set object on the right side.

```
bool operator==(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the set on the left side of the operator is equal to the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// set_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 == s2 )
        cout << "The sets s1 and s2 are equal." << endl;
    else
        cout << "The sets s1 and s2 are not equal." << endl;

    if ( s1 == s3 )
        cout << "The sets s1 and s3 are equal." << endl;
    else
        cout << "The sets s1 and s3 are not equal." << endl;
}
/* Output:
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
*/
```

operator> (set)

Tests if the set object on the left side of the operator is greater than the set object on the right side.

```
bool operator>(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the set on the left side of the operator is greater than the set on the right side of the operator; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// set_op_gt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
    }

    if ( s1 > s2 )
        cout << "The set s1 is greater than the set s2." << endl;
    else
        cout << "The set s1 is not greater than the set s2." << endl;

    if ( s1 > s3 )
        cout << "The set s1 is greater than the set s3." << endl;
    else
        cout << "The set s1 is not greater than the set s3." << endl;
}
/* Output:
The set s1 is not greater than the set s2.
The set s1 is greater than the set s3.
*/

```

operator>= (set)

Tests if the set object on the left side of the operator is greater than or equal to the set object on the right side.

```
bool operator!>=(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right

An object of type `set`.

Return Value

true if the set on the left side of the operator is greater than or equal to the set on the right side of the list; otherwise **false**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example


```

// set_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3, s4;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
        s4.insert ( i );
    }

    if ( s1 >= s2 )
        cout << "Set s1 is greater than or equal to set s2." << endl;
    else
        cout << "The set s1 is less than the set s2." << endl;

    if ( s1 >= s3 )
        cout << "Set s1 is greater than or equal to set s3." << endl;
    else
        cout << "The set s1 is less than the set s3." << endl;

    if ( s1 >= s4 )
        cout << "Set s1 is greater than or equal to set s4." << endl;
    else
        cout << "The set s1 is less than the set s4." << endl;
}
/* Output:
The set s1 is less than the set s2.
Set s1 is greater than or equal to set s3.
Set s1 is greater than or equal to set s4.
*/

```

operator!= (multiset)

Tests if the multiset object on the left side of the operator is not equal to the multiset object on the right side.

```
bool operator!=(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `multiset`.

right

An object of type `multiset`.

Return Value

true if the sets or multisets are not equal; **false** if sets or multisets are equal.

Remarks

The comparison between multiset objects is based on a pairwise comparison between their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multiset_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 != s2 )
        cout << "The multisets s1 and s2 are not equal." << endl;
    else
        cout << "The multisets s1 and s2 are equal." << endl;

    if ( s1 != s3 )
        cout << "The multisets s1 and s3 are not equal." << endl;
    else
        cout << "The multisets s1 and s3 are equal." << endl;
}
/* Output:
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
*/
```

operator< (multiset)

Tests if the multiset object on the left side of the operator is less than the multiset object on the right side.

```
bool operator<(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `multiset`.

right

An object of type `multiset`.

Return Value

true if the multiset on the left side of the operator is strictly less than the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```

// multiset_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
    }

    if ( s1 < s2 )
        cout << "The multiset s1 is less than "
              << "the multiset s2." << endl;
    else
        cout << "The multiset s1 is not less than "
              << "the multiset s2." << endl;

    if ( s1 < s3 )
        cout << "The multiset s1 is less than "
              << "the multiset s3." << endl;
    else
        cout << "The multiset s1 is not less than "
              << "the multiset s3." << endl;
}
/* Output:
The multiset s1 is less than the multiset s2.
The multiset s1 is not less than the multiset s3.
*/

```

operator<= (multiset)

Tests if the multiset object on the left side of the operator is less than or equal to the multiset object on the right side.

```

bool operator!<=(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>&
right);

```

Parameters

left

An object of type `multiset`.

right

An object of type `multiset`.

Return Value

true if the multiset on the left side of the operator is less than or equal to the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3, s4;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
        s4.insert ( i );
    }

    if ( s1 <= s2 )
        cout << "The multiset s1 is less than "
              << "or equal to the multiset s2." << endl;
    else
        cout << "The multiset s1 is greater than "
              << "the multiset s2." << endl;

    if ( s1 <= s3 )
        cout << "The multiset s1 is less than "
              << "or equal to the multiset s3." << endl;
    else
        cout << "The multiset s1 is greater than "
              << "the multiset s3." << endl;

    if ( s1 <= s4 )
        cout << "The multiset s1 is less than "
              << "or equal to the multiset s4." << endl;
    else
        cout << "The multiset s1 is greater than "
              << "the multiset s4." << endl;
}
/* Output:
The multiset s1 is less than or equal to the multiset s2.
The multiset s1 is greater than the multiset s3.
The multiset s1 is less than or equal to the multiset s4.
*/
```

operator== (multiset)

Tests if the multiset object on the left side of the operator is equal to the multiset object on the right side.

```
bool operator!=(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>&
right);
```

Parameters

left

An object of type `multiset`.

right

An object of type `multiset`.

Return Value

true if the multiset on the left side of the operator is equal to the multiset on the right side of the operator; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// multiset_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i );
    }

    if ( s1 == s2 )
        cout << "The multisets s1 and s2 are equal." << endl;
    else
        cout << "The multisets s1 and s2 are not equal." << endl;

    if ( s1 == s3 )
        cout << "The multisets s1 and s3 are equal." << endl;
    else
        cout << "The multisets s1 and s3 are not equal." << endl;
}
/* Output:
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
*/
```

operator> (multiset)

Tests if the multiset object on the left side of the operator is greater than the multiset object on the right side.

```
bool operator>(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>& right);
```

Parameters

left

An object of type `multiset`.

right

An object of type `multiset`.

Return Value

true if the multiset on the left side of the operator is greater than the multiset on the right side of the operator;

otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_gt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
    }

    if ( s1 > s2 )
        cout << "The multiset s1 is greater than "
              << "the multiset s2." << endl;
    else
        cout << "The multiset s1 is not greater "
              << "than the multiset s2." << endl;

    if ( s1 > s3 )
        cout << "The multiset s1 is greater than "
              << "the multiset s3." << endl;
    else
        cout << "The multiset s1 is not greater than "
              << "the multiset s3." << endl;
}
/* Output:
The multiset s1 is not greater than the multiset s2.
The multiset s1 is greater than the multiset s3.
*/
```

operator>= (multiset)

Tests if the multiset object on the left side of the operator is greater than or equal to the multiset object on the right side.

```
bool operator!=(const multiset <Key, Traits, Allocator>& left, const multiset <Key, Traits, Allocator>&
right);
```

Parameters

left

An object of type `multiset` .

right

An object of type `multiset` .

Return Value

true if the multiset on the left side of the operator is greater than or equal to the multiset on the right side of the list; otherwise **false**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
// multiset_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> s1, s2, s3, s4;
    int i;

    for ( i = 0 ; i < 3 ; i++ )
    {
        s1.insert ( i );
        s2.insert ( i * i );
        s3.insert ( i - 1 );
        s4.insert ( i );
    }

    if ( s1 >= s2 )
        cout << "The multiset s1 is greater than "
              << "or equal to the multiset s2." << endl;
    else
        cout << "The multiset s1 is less than "
              << "the multiset s2." << endl;

    if ( s1 >= s3 )
        cout << "The multiset s1 is greater than "
              << "or equal to the multiset s3." << endl;
    else
        cout << "The multiset s1 is less than "
              << "the multiset s3." << endl;

    if ( s1 >= s4 )
        cout << "The multiset s1 is greater than "
              << "or equal to the multiset s4." << endl;
    else
        cout << "The multiset s1 is less than "
              << "the multiset s4." << endl;
}
/* Output:
The multiset s1 is less than the multiset s2.
The multiset s1 is greater than or equal to the multiset s3.
The multiset s1 is greater than or equal to the multiset s4.
*/
```

See also

[<set>](#)

set Class

11/15/2018 • 51 minutes to read • [Edit Online](#)

The C++ Standard Library container class `set` is used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

Syntax

```
template <class Key,  
         class Traits=less<Key>,  
         class Allocator=allocator<Key>>  
class set
```

Parameters

Key

The element data type to be stored in the set.

Traits

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the set. This argument is optional, and the binary predicate **less** <Key> is the default value.

In C++ 14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Allocator

The type that represents the stored allocator object that encapsulates details about the set's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

A C++ Standard Library `set` is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Unique in the sense that each of its elements must have a unique key. Since `set` is also a simple associative container, its elements are also unique.

A `set` is also described as a template class because the functionality it provides is generic and independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The

member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a set are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure. If values need to be attached to a list of unique key words, then a map would be an appropriate structure to contain this data. If instead the keys are not unique, then a multimap would be the container of choice.

The set orders the sequence it controls by calling a stored function object of type [key_compare](#). This stored object is a comparison function that may be accessed by calling the member function [key_comp](#). In general, the elements need to be merely less than comparable to establish this order so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x,y)$ is a function object that has two argument objects x and y and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both $f(x,y)$ and $f(y,x)$ are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++ 14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

The iterator provided by the set class is a bidirectional iterator, but the class member functions [insert](#) and [set](#) have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`First` , `Last`) in the context of the class's member functions.

Constructors

CONSTRUCTOR	DESCRIPTION
set	Constructs a set that is empty or that is a copy of all or part of some other set.

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	A type that represents the <code>allocator</code> class for the set object.
const_iterator	A type that provides a bidirectional iterator that can read a const element in the set.
const_pointer	A type that provides a pointer to a const element in a set.

TYPE NAME	DESCRIPTION
<code>const_reference</code>	A type that provides a reference to a const element stored in a set for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any const element in the set.
<code>difference_type</code>	A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.
<code>iterator</code>	A type that provides a bidirectional iterator that can read or modify any element in a set.
<code>key_compare</code>	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.
<code>key_type</code>	The type describes an object stored as an element of a set in its capacity as sort key.
<code>pointer</code>	A type that provides a pointer to an element in a set.
<code>reference</code>	A type that provides a reference to an element stored in a set.
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed set.
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a set.
<code>value_compare</code>	The type that provides a function object that can compare two elements to determine their relative order in the set.
<code>value_type</code>	The type describes an object stored as an element of a set in its capacity as a value.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns an iterator that addresses the first element in the set.
<code>cbegin</code>	Returns a const iterator that addresses the first element in the set.
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a set.
<code>clear</code>	Erases all the elements of a set.
<code>count</code>	Returns the number of elements in a set whose key matches a parameter-specified key.

MEMBER FUNCTION	DESCRIPTION
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed set.
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed set.
<code>emplace</code>	Inserts an element constructed in place into a set.
<code>emplace_hint</code>	Inserts an element constructed in place into a set, with a placement hint.
<code>empty</code>	Tests if a set is empty.
<code>end</code>	Returns an iterator that addresses the location succeeding the last element in a set.
<code>equal_range</code>	Returns a pair of iterators respectively to the first element in a set with a key that is greater than a specified key and to the first element in the set with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.
<code>find</code>	Returns an iterator addressing the location of an element in a set that has a key equivalent to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct the set.
<code>insert</code>	Inserts an element or a range of elements into a set.
<code>key_comp</code>	Retrieves a copy of the comparison object used to order keys in a set.
<code>lower_bound</code>	Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.
<code>max_size</code>	Returns the maximum length of the set.
<code>rbegin</code>	Returns an iterator addressing the first element in a reversed set.
<code>rend</code>	Returns an iterator that addresses the location succeeding the last element in a reversed set.
<code>size</code>	Returns the number of elements in the set.
<code>swap</code>	Exchanges the elements of two sets.
<code>upper_bound</code>	Returns an iterator to the first element in a set with a key that is greater than a specified key.

MEMBER FUNCTION	DESCRIPTION
value_comp	Retrieves a copy of the comparison object used to order element values in a set.

Operators

OPERATOR	DESCRIPTION
operator=	Replaces the elements of a set with a copy of another set.

Requirements

Header: <set>

Namespace: std

set::allocator_type

A type that represents the allocator class for the set object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter [Allocator](#).

Returns the function object that a multiset uses to order its elements, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the [set Class](#) topic.

Example

See the example for [get_allocator](#) for an example that uses `allocator_type`.

set::begin

Returns an iterator that addresses the first element in the set.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the set or the location succeeding an empty set.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the set object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the set object can be modified.

Example

```

// set_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int>::iterator s1_Iter;
    set <int>::const_iterator s1_cIter;

    s1.insert( 1 );
    s1.insert( 2 );
    s1.insert( 3 );

    s1_Iter = s1.begin( );
    cout << "The first element of s1 is " << *s1_Iter << endl;

    s1_Iter = s1.begin( );
    s1.erase( s1_Iter );

    // The following 2 lines would err because the iterator is const
    // s1_cIter = s1.begin( );
    // s1.erase( s1_cIter );

    s1_cIter = s1.begin( );
    cout << "The first element of s1 is now " << *s1_cIter << endl;
}

```

```

The first element of s1 is 1
The first element of s1 is now 2

```

set::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the [auto](#) type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```

auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator

```

set::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

set::clear

Erases all the elements of a set.

```
void clear();
```

Example

```
// set_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;

    s1.insert( 1 );
    s1.insert( 2 );

    cout << "The size of the set is initially " << s1.size( )
         << "." << endl;

    s1.clear( );
    cout << "The size of the set after clearing is "
         << s1.size( ) << "." << endl;
}
```

The size of the set is initially 2.
The size of the set after clearing is 0.

set::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the set.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for [begin](#) for an example that uses `const_iterator`.

set::const_pointer

A type that provides a pointer to a **const** element in a set.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, a [const_iterator](#) should be used to access the elements in a const set object.

set::const_reference

A type that provides a reference to a **const** element stored in a set for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// set_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;

    s1.insert( 10 );
    s1.insert( 20 );

    // Declare and initialize a const_reference &Ref1
    // to the 1st element
    const int &Ref1 = *s1.begin( );

    cout << "The first element in the set is "
         << Ref1 << "." << endl;

    // The following line would cause an error because the
    // const_reference cannot be used to modify the set
    // Ref1 = Ref1 + 5;
}
```

The first element in the set is 10.

set::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the set.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the set in reverse.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`.

set::count

Returns the number of elements in a set whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the set.

Return Value

1 if the set contains an element whose sort key matches the parameter key. 0 if the set does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[lower_bound(key), upper_bound(key)).

Example

The following example demonstrates the use of the set::count member function.

```
// set_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    set<int> s1;
    set<int>::size_type i;

    s1.insert(1);
    s1.insert(1);

    // Keys must be unique in set, so duplicates are ignored
    i = s1.count(1);
    cout << "The number of elements in s1 with a sort key of 1 is: "
         << i << "." << endl;

    i = s1.count(2);
    cout << "The number of elements in s1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

```
The number of elements in s1 with a sort key of 1 is: 1.
The number of elements in s1 with a sort key of 2 is: 0.
```

set::crbegin

Returns a const iterator addressing the first element in a reversed set.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

`crbegin` is used with a reversed set just as `begin` is used with a set.

With the return value of `crbegin`, the set object cannot be modified.

Example

```
// set_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int>::const_reverse_iterator s1_crIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_crIter = s1.crbegin( );
    cout << "The first element in the reversed set is "
         << *s1_crIter << "." << endl;
}
```

```
The first element in the reversed set is 30.
```

set::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed set.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed set (the location that had preceded the first element in the unreversed set).

Remarks

`crend` is used with a reversed set just as `end` is used with a set.

With the return value of `crend`, the set object cannot be modified. The value returned by `crend` should not be dereferenced.

`crend` can be used to test to whether a reverse iterator has reached the end of its set.

Example

```

// set_crend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
    using namespace std;
    set<int> s1;
    set<int>::const_reverse_iterator s1_crIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_crIter = s1.crend( );
    s1_crIter--;
    cout << "The last element in the reversed set is "
         << *s1_crIter << "." << endl;
}

```

set::difference_type

A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range $[first, last)$ between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as `set`, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as `vector`.

Example

```

// set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
    using namespace std;

    set <int> s1;
    set <int>::iterator s1_iter, s1_bIter, s1_eIter;

    s1.insert( 20 );
    s1.insert( 10 );
    s1.insert( 20 );    // won't insert as set elements are unique

    s1_bIter = s1.begin( );
    s1_eIter = s1.end( );

    set <int>::difference_type  df_typ5, df_typ10, df_typ20;

    df_typ5 = count( s1_bIter, s1_eIter, 5 );
    df_typ10 = count( s1_bIter, s1_eIter, 10 );
    df_typ20 = count( s1_bIter, s1_eIter, 20 );

    // the keys, and hence the elements of a set are unique,
    // so there is at most one of a given value
    cout << "The number '5' occurs " << df_typ5
         << " times in set s1.\n";
    cout << "The number '10' occurs " << df_typ10
         << " times in set s1.\n";
    cout << "The number '20' occurs " << df_typ20
         << " times in set s1.\n";

    // count the number of elements in a set
    set <int>::difference_type  df_count = 0;
    s1_iter = s1.begin( );
    while ( s1_iter != s1_eIter)
    {
        df_count++;
        s1_iter++;
    }

    cout << "The number of elements in the set s1 is: "
         << df_count << "." << endl;
}

```

```

The number '5' occurs 0 times in set s1.
The number '10' occurs 1 times in set s1.
The number '20' occurs 1 times in set s1.
The number of elements in the set s1 is: 2.

```

set::emplace

Inserts an element constructed in place (no copy or move operations are performed).

```

template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.

Return Value

A [pair](#) whose bool component returns true if an insertion was made, and false if the map already contained an element whose value had an equivalent value in the ordering. The iterator component of the return value pair returns the address where a new element was inserted (if the bool component is true) or where the element was already located (if the bool component is false).

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

Example

```

// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    set<string> s1;

    auto ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
              << endl << " The existing element is (" << *ret.first << ")"
              << endl;
        cout << "set not modified" << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;

    ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
              << endl << " The existing element is (" << *ret.first << ")"
              << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;
}

```

set::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```

template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the set unless the set already contains that element or, more generally, unless it already contains an element whose value is equivalently ordered.
<i>where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>where</i> , insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

Example

```
// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: " << endl;

    for (const auto& p : s) {
        cout << "(" << p << " ) ";
    }

    cout << endl;
}

int main()
{
    set<string> s1;

    // Emplace some test data
    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmine");

    cout << "set starting data: ";
    print(s1);
    cout << endl;

    // Emplace with hint
    // s1.end() should be the "next" element after this emplacement
    s1.emplace_hint(s1.end(), "Doug");

    cout << "set modified, now contains ";
    print(s1);
    cout << endl;
}
```

set::empty

Tests if a set is empty.

```
bool empty() const;
```

Return Value

true if the set is empty; **false** if the set is nonempty.

Example

```
// set_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2;
    s1.insert ( 1 );

    if ( s1.empty( ) )
        cout << "The set s1 is empty." << endl;
    else
        cout << "The set s1 is not empty." << endl;

    if ( s2.empty( ) )
        cout << "The set s2 is empty." << endl;
    else
        cout << "The set s2 is not empty." << endl;
}
```

```
The set s1 is not empty.
The set s2 is empty.
```

set::end

Returns the past-the-end iterator.

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the set is empty, then `set::end() == set::begin()`.

Remarks

end is used to test whether an iterator has passed the end of its set.

The value returned by **end** should not be dereferenced.

For a code example, see [set::find](#).

set::equal_range

Returns a pair of iterators respectively to the first element in a set with a key that is greater than or equal to a

specified key and to the first element in the set with a key that is greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;  
  
pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

A pair of iterators where the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first`, and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second`, and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```

// set_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    typedef set<int, less< int > > IntSet;
    IntSet s1;
    set<int, less< int > > :: const_iterator s1_RcIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    pair<IntSet::const_iterator, IntSet::const_iterator> p1, p2;
    p1 = s1.equal_range( 20 );

    cout << "The upper bound of the element with "
         << "a key of 20 in the set s1 is: "
         << *(p1.second) << "." << endl;

    cout << "The lower bound of the element with "
         << "a key of 20 in the set s1 is: "
         << *(p1.first) << "." << endl;

    // Compare the upper_bound called directly
    s1_RcIter = s1.upper_bound( 20 );
    cout << "A direct call of upper_bound( 20 ) gives "
         << *s1_RcIter << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

    p2 = s1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == s1.end( ) ) && ( p2.second == s1.end( ) ) )
        cout << "The set s1 doesn't have an element "
             << "with a key less than 40." << endl;
    else
        cout << "The element of set s1 with a key >= 40 is: "
             << *(p1.first) << "." << endl;
}

```

```

The upper bound of the element with a key of 20 in the set s1 is: 30.
The lower bound of the element with a key of 20 in the set s1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The set s1 doesn't have an element with a key less than 40.

```

set::erase

Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.

```

iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);

```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the set if no such element exists.

For the third member function, returns the number of elements that have been removed from the set.

Remarks

Example

```

// set_erase.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions

using namespace std;

using myset = set<string>;

void printset(const myset& s) {
    for (const auto& iter : s) {
        cout << " [" << iter << "]\n";
    }
    cout << endl << "size() == " << s.size() << endl << endl;
}

int main()
{
    myset s1;

    // Fill in some data to test with, one at a time
    s1.insert("Bob");
    s1.insert("Robert");
    s1.insert("Bert");
    s1.insert("Rob");
    s1.insert("Bobby");

    cout << "Starting data of set s1 is:" << endl;
}

```

```

printset(s1);
// The 1st member function removes an element at a given position
s1.erase(next(s1.begin()));
cout << "After the 2nd element is deleted, the set s1 is:" << endl;
printset(s1);

// Fill in some data to test with, one at a time, using an initializer list
myset s2{ "meow", "hiss", "purr", "growl", "yowl" };

cout << "Starting data of set s2 is:" << endl;
printset(s2);
// The 2nd member function removes elements
// in the range [First, Last)
s2.erase(next(s2.begin()), prev(s2.end()));
cout << "After the middle elements are deleted, the set s2 is:" << endl;
printset(s2);

myset s3;

// Fill in some data to test with, one at a time, using emplace
s3.emplace("C");
s3.emplace("C#");
s3.emplace("D");
s3.emplace("D#");
s3.emplace("E");
s3.emplace("E#");
s3.emplace("F");
s3.emplace("F#");
s3.emplace("G");
s3.emplace("G#");
s3.emplace("A");
s3.emplace("A#");
s3.emplace("B");

cout << "Starting data of set s3 is:" << endl;
printset(s3);
// The 3rd member function removes elements with a given Key
myset::size_type count = s3.erase("E#");
// The 3rd member function also returns the number of elements removed
cout << "The number of elements removed from s3 is: " << count << "." << endl;
cout << "After the element with a key of \"E#\" is deleted, the set s3 is:" << endl;
printset(s3);
}

```

set::find

Returns an iterator that refers to the location of an element in a set that has a key equivalent to a specified key.

```

iterator find(const Key& key);

const_iterator find(const Key& key) const;

```

Parameters

key

The key value to be matched by the sort key of an element from the set being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the set (`set::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the set whose key is equivalent to the

argument *key* under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the set object cannot be modified. If the return value of `find` is assigned to an `iterator`, the set object can be modified

Example

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    set<int> s1({ 40, 45 });
    cout << "The starting set s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate

    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified set s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

`set::get_allocator`

Returns a copy of the allocator object used to construct the set.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the set to manage memory, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the [set Class](#) topic.

Remarks

Allocators for the set class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// set_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int>::allocator_type s1_Alloc;
    set <int>::allocator_type s2_Alloc;
    set <double>::allocator_type s3_Alloc;
    set <int>::allocator_type s4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    set <int> s1;
    set <int, allocator<int> > s2;
    set <double, allocator<double> > s3;

    s1_Alloc = s1.get_allocator( );
    s2_Alloc = s2.get_allocator( );
    s3_Alloc = s3.get_allocator( );

    cout << "The number of integers that can be allocated"
         << endl << "before free memory is exhausted: "
         << s2.max_size( ) << "." << endl;

    cout << "\nThe number of doubles that can be allocated"
         << endl << "before free memory is exhausted: "
         << s3.max_size( ) << "." << endl;

    // The following line creates a set s4
    // with the allocator of multiset s1.
    set <int> s4( less<int>( ), s1_Alloc );

    s4_Alloc = s4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated by the other
    if( s1_Alloc == s4_Alloc )
    {
        cout << "\nThe allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "\nThe allocators are not interchangeable."
             << endl;
    }
}

```

set::insert

Inserts an element or a range of elements into a set.

```

// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);

```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>Where</i> , insertion can occur in amortized constant time instead of logarithmic time.)
<i>ValTy</i>	Template parameter that specifies the argument type that the set can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a [pair](#) whose **bool** component is true if an insertion was

made, and false if the set already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is true, or to the existing element if the **bool** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the set or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the **bool** component, use `pr.second`. For an example, see the sample code later in this article.

The [value_type](#) of a container is a typedef that belongs to the container, and, for set, `set<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a set that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `s.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `s`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the set.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [set::emplace](#) and [set::emplace_hint](#).

Example

```
// set_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    // insert single values
    set<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
```

```

s1.insert(20);

cout << "The original set values of s1 are:" << endl;
print(s1);

// intentionally attempt a duplicate, single element
auto ret = s1.insert(1);
if (!ret.second){
    auto elem = *ret.first;
    cout << "Insert failed, element with value 1 already exists."
        << endl << " The existing element is (" << elem << ")"
        << endl;
}
else{
    cout << "The modified set values of s1 are:" << endl;
    print(s1);
}
cout << endl;

// single element, with hint
s1.insert(s1.end(), 30);
cout << "The modified set values of s1 are:" << endl;
print(s1);
cout << endl;

// The templated version inserting a jumbled range
set<int> s2;
vector<int> v;
v.push_back(43);
v.push_back(294);
v.push_back(41);
v.push_back(330);
v.push_back(42);
v.push_back(45);

cout << "Inserting the following vector data into s2:" << endl;
print(v);

s2.insert(v.begin(), v.end());

cout << "The modified set values of s2 are:" << endl;
print(s2);
cout << endl;

// The templated versions move-constructing elements
set<string> s3;
string str1("blue"), str2("green");

// single element
s3.insert(move(str1));
cout << "After the first move insertion, s3 contains:" << endl;
print(s3);

// single element with hint
s3.insert(s3.end(), move(str2));
cout << "After the second move insertion, s3 contains:" << endl;
print(s3);
cout << endl;

set<int> s4;
// Insert the elements from an initializer_list
s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
cout << "After initializer_list insertion, s4 contains:" << endl;
print(s4);
cout << endl;
}

```

set::iterator

A type that provides a constant [bidirectional iterator](#) that can read any element in a set.

```
typedef implementation-defined iterator;
```

Example

See the example for [begin](#) for an example of how to declare and use an `iterator`.

set::key_comp

Retrieves a copy of the comparison object used to order keys in a set.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits` see the [set Class](#) topic.

Remarks

The stored object defines the member function:

```
bool operator()( const Key& _xVal, const Key& _yVal );
```

which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both [key_compare](#) and [value_compare](#) are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```

// set_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;

    set <int, less<int> > s1;
    set<int, less<int> >::key_compare kc1 = s1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
              << "where kc1 is the function object of s1."
              << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
              << "where kc1 is the function object of s1."
              << endl;
    }

    set <int, greater<int> > s2;
    set<int, greater<int> >::key_compare kc2 = s2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if(result2 == true)
    {
        cout << "kc2( 2,3 ) returns value of true, "
              << "where kc2 is the function object of s2."
              << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
              << "where kc2 is the function object of s2."
              << endl;
    }
}

```

```

kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of s2.

```

set::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the [set Class](#) topic.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for [key_comp](#) for an example of how to declare and use `key_compare`.

set::key_type

A type that describes an object stored as an element of a set in its capacity as sort key.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `key`, see the Remarks section of the [set Class](#) topic.

Note that both `key_type` and [value_type](#) are synonyms for the template parameter `Key`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for [value_type](#) for an example of how to declare and use `key_type`.

set::lower_bound

Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(const Key& key) const;  
  
iterator lower_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```

// set_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int> :: const_iterator s1_AcIter, s1_RcIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_RcIter = s1.lower_bound( 20 );
    cout << "The element of set s1 with a key of 20 is: "
         << *s1_RcIter << "." << endl;

    s1_RcIter = s1.lower_bound( 40 );

    // If no match is found for the key, end( ) is returned
    if ( s1_RcIter == s1.end( ) )
        cout << "The set s1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of set s1 with a key of 40 is: "
             << *s1_RcIter << "." << endl;

    // The element at a specific location in the set can be found
    // by using a dereferenced iterator that addresses the location
    s1_AcIter = s1.end( );
    s1_AcIter--;
    s1_RcIter = s1.lower_bound( *s1_AcIter );
    cout << "The element of s1 with a key matching "
         << "that of the last element is: "
         << *s1_RcIter << "." << endl;
}

```

```

The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.

```

set::max_size

Returns the maximum length of the set.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the set.

Example

```
// set_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int>::size_type i;

    i = s1.max_size( );
    cout << "The maximum possible length "
         << "of the set is " << i << "." << endl;
}
```

set::operator=

Replaces the elements of this `set` using elements from another `set`.

```
set& operator=(const set& right);

set& operator=(set&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>set</code> providing new elements to be assigned to this <code>set</code> .

Remarks

The first version of `operator=` uses an [lvalue reference](#) for *right*, to copy elements from *right* to this `set`.

The second version uses an [rvalue reference](#) for *right*. It moves elements from *right* to this `set`.

Any elements in this `set` before the operator function executes are discarded.

Example

```

// set_operator_as.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set<int> v1, v2, v3;
    set<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

```

set::pointer

A type that provides a pointer to an element in a set.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a set object.

set::rbegin

Returns an iterator addressing the first element in a reversed set.

```

const_reverse_iterator rbegin() const;

reverse_iterator rbegin();

```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

`rbegin` is used with a reversed set just as [begin](#) is used with a set.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the set object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the set object can be modified.

`rbegin` can be used to iterate through a set backwards.

Example

```
// set_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int>::iterator s1_Iter;
    set <int>::reverse_iterator s1_rIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_rIter = s1.rbegin( );
    cout << "The first element in the reversed set is "
         << *s1_rIter << "." << endl;

    // begin can be used to start an iteration
    // through a set in a forward order
    cout << "The set is:";
    for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ) ; s1_Iter++ )
        cout << " " << *s1_Iter;
    cout << endl;

    // rbegin can be used to start an iteration
    // through a set in a reverse order
    cout << "The reversed set is:";
    for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ) ; s1_rIter++ )
        cout << " " << *s1_rIter;
    cout << endl;

    // A set element can be erased by dereferencing to its key
    s1_rIter = s1.rbegin( );
    s1.erase ( *s1_rIter );

    s1_rIter = s1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed set is "<< *s1_rIter << "." << endl;
}
```

```
The first element in the reversed set is 30.
The set is: 10 20 30
The reversed set is: 30 20 10
After the erasure, the first element in the reversed set is 20.
```

set::reference

A type that provides a reference to an element stored in a set.

```
typedef typename allocator_type::reference reference;
```

Example

```
// set_reference.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;

    s1.insert( 10 );
    s1.insert( 20 );

    // Declare and initialize a reference &Ref1 to the 1st element
    const int &Ref1 = *s1.begin( );

    cout << "The first element in the set is "
         << Ref1 << "." << endl;
}
```

```
The first element in the set is 10.
```

set::rend

Returns an iterator that addresses the location succeeding the last element in a reversed set.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed set (the location that had preceded the first element in the unreversed set).

Remarks

`rend` is used with a reversed set just as `end` is used with a set.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the set object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the set object can be modified. The value returned by `rend` should not be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its set.

Example

```

// set_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
    using namespace std;
    set <int> s1;
    set <int>::iterator s1_Iter;
    set <int>::reverse_iterator s1_rIter;
    set <int>::const_reverse_iterator s1_crIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_rIter = s1.rend( );
    s1_rIter--;
    cout << "The last element in the reversed set is "
         << *s1_rIter << "." << endl;

    // end can be used to terminate an iteration
    // through a set in a forward order
    cout << "The set is: ";
    for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ) ; s1_Iter++ )
        cout << *s1_Iter << " ";
    cout << "." << endl;

    // rend can be used to terminate an iteration
    // through a set in a reverse order
    cout << "The reversed set is: ";
    for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ) ; s1_rIter++ )
        cout << *s1_rIter << " ";
    cout << "." << endl;

    s1_rIter = s1.rend( );
    s1_rIter--;
    s1.erase ( *s1_rIter );

    s1_rIter = s1.rend( );
    --s1_rIter;
    cout << "After the erasure, the last element in the "
         << "reversed set is " << *s1_rIter << "." << endl;
}

```

set::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed set.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the set in reverse.

Example

See the example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

set::set

Constructs a set that is empty or that is a copy of all or part of some other set.

```

set();

explicit set(
    const Traits& Comp);

set(
    const Traits& Comp,
    const Allocator& Al);

set(
    const set& Right);

set(
    set&& Right);

set(
    initializer_list<Type> IList);

set(
    initializer_list<Type> IList,
    const Compare& Comp);

set(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this set object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Traits</code> used to order the elements in the set, which defaults to <code>Compare</code> .
<i>Right</i>	The set of which the constructed set is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.
<i>Last</i>	The position of the first element beyond the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the set and that can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their sets.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the set and that can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial set, the second specifying the type of comparison function (`comp`) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (`al`) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the set `right`.

The next three constructors use an initializer_list to specify the elements.

The next three constructors copy the range [`first`, `last`) of a set with increasing explicitness in specifying the type of comparison function of class `Traits` and **Allocator**.

The eighth constructor specifies a copy of the set by moving `right`.

Example

```
// set_set.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;

    // Create an empty set s0 of key type integer
    set<int> s0;

    // Create an empty set s1 with the key comparison
    // function of less than, then insert 4 elements
    set<int, less<int> > s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(30);
    s1.insert(40);

    // Create an empty set s2 with the key comparison
    // function of less than, then insert 2 elements
    set<int, less<int> > s2;
    s2.insert(10);
    s2.insert(20);

    // Create a set s3 with the
    // allocator of set s1
    set<int::allocator_type s1_Alloc;
    s1_Alloc = s1.get_allocator();
    set<int> s3(less<int>(), s1_Alloc);
    s3.insert(30);

    // Create a copy, set s4, of set s1
```

```

set<int> s4(s1);

// Create a set s5 by copying the range s1[ first, last)
set<int>::const_iterator s1_bcIter, s1_ecIter;
s1_bcIter = s1.begin();
s1_ecIter = s1.begin();
s1_ecIter++;
s1_ecIter++;
set<int> s5(s1_bcIter, s1_ecIter);

// Create a set s6 by copying the range s4[ first, last)
// and with the allocator of set s2
set<int>::allocator_type s2_Alloc;
s2_Alloc = s2.get_allocator();
set<int> s6(s4.begin(), ++s4.begin(), less<int>(), s2_Alloc);

cout << "s1 =";
for (auto i : s1)
    cout << " " << i;
cout << endl;

cout << "s2 = " << *s2.begin() << " " << *++s2.begin() << endl;

cout << "s3 =";
for (auto i : s3)
    cout << " " << i;
cout << endl;

cout << "s4 =";
for (auto i : s4)
    cout << " " << i;
cout << endl;

cout << "s5 =";
for (auto i : s5)
    cout << " " << i;
cout << endl;

cout << "s6 =";
for (auto i : s6)
    cout << " " << i;
cout << endl;

// Create a set by moving s5
set<int> s7(move(s5));
cout << "s7 =";
for (auto i : s7)
    cout << " " << i;
cout << endl;

// Create a set with an initializer_list
cout << "s8 =";
set<int> s8{ { 1, 2, 3, 4 } };
for (auto i : s8)
    cout << " " << i;
cout << endl;

cout << "s9 =";
set<int> s9{ { 5, 6, 7, 8 }, less<int>() };
for (auto i : s9)
    cout << " " << i;
cout << endl;

cout << "s10 =";
set<int> s10{ { 10, 20, 30, 40 }, less<int>(), s9.get_allocator() };
for (auto i : s10)
    cout << " " << i;
cout << endl;
}

```

```
s1 = 10 20 30 40 s2 = 10 20 s3 = 30 s4 = 10 20 30 40 s5 = 10 20 s6 = 10 s7 = 10 20 s8 = 1 2 3 4 s9 = 5 6 7 8 s10 = 10 20 30 40
```

set::size

Returns the number of elements in the set.

```
size_type size() const;
```

Return Value

The current length of the set.

Example

```
// set_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int> :: size_type i;

    s1.insert( 1 );
    i = s1.size( );
    cout << "The set length is " << i << "." << endl;

    s1.insert( 2 );
    i = s1.size( );
    cout << "The set length is now " << i << "." << endl;
}
```

```
The set length is 1.
The set length is now 2.
```

set::size_type

An unsigned integer type that can represent the number of elements in a set.

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for [size](#) for an example of how to declare and use `size_type`

set::swap

Exchanges the elements of two sets.

```
void swap(
    set<Key, Traits, Allocator>& right);
```

Parameters

right

The argument set providing the elements to be swapped with the target set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two sets whose elements are being exchanged.

Example

```
// set_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1, s2, s3;
    set <int>::iterator s1_Iter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );
    s2.insert( 100 );
    s2.insert( 200 );
    s3.insert( 300 );

    cout << "The original set s1 is:";
    for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter;
    cout << "." << endl;

    // This is the member function version of swap
    s1.swap( s2 );

    cout << "After swapping with s2, list s1 is:";
    for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( s1, s3 );

    cout << "After swapping with s3, list s1 is:";
    for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter;
    cout << "." << endl;
}
```

```
The original set s1 is: 10 20 30.
After swapping with s2, list s1 is: 100 200.
After swapping with s3, list s1 is: 300.
```

set::upper_bound

Returns an iterator to the first element in a set that with a key that is greater than a specified key.


```
const_iterator upper_bound(const Key& key) const;
```

```
iterator upper_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a set that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```
// set_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int> :: const_iterator s1_AcIter, s1_RcIter;

    s1.insert( 10 );
    s1.insert( 20 );
    s1.insert( 30 );

    s1_RcIter = s1.upper_bound( 20 );
    cout << "The first element of set s1 with a key greater "
          << "than 20 is: " << *s1_RcIter << "." << endl;

    s1_RcIter = s1.upper_bound( 30 );

    // If no match is found for the key, end( ) is returned
    if ( s1_RcIter == s1.end( ) )
        cout << "The set s1 doesn't have an element "
              << "with a key greater than 30." << endl;
    else
        cout << "The element of set s1 with a key > 40 is: "
              << *s1_RcIter << "." << endl;

    // The element at a specific location in the set can be found
    // by using a dereferenced iterator addressing the location
    s1_AcIter = s1.begin( );
    s1_RcIter = s1.upper_bound( *s1_AcIter );
    cout << "The first element of s1 with a key greater than"
          << endl << "that of the initial element of s1 is: "
          << *s1_RcIter << "." << endl;
}
```

The first element of set s1 with a key greater than 20 is: 30.
The set s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than
that of the initial element of s1 is: 20.

set::value_comp

Retrieves a copy of the comparison object used to order element values in a set.

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits` see the [set Class](#) topic.

Remarks

The stored object defines the member function:

```
bool operator( const Key& _xVal , const Key& _yVal );
```

which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both `value_compare` and `key_compare` are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```

// set_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;

    set <int, less<int> > s1;
    set <int, less<int> >::value_compare vc1 = s1.value_comp( );
    bool result1 = vc1( 2, 3 );
    if( result1 == true )
    {
        cout << "vc1( 2,3 ) returns value of true, "
              << "where vc1 is the function object of s1."
              << endl;
    }
    else
    {
        cout << "vc1( 2,3 ) returns value of false, "
              << "where vc1 is the function object of s1."
              << endl;
    }

    set <int, greater<int> > s2;
    set<int, greater<int> >::value_compare vc2 = s2.value_comp( );
    bool result2 = vc2( 2, 3 );
    if( result2 == true )
    {
        cout << "vc2( 2,3 ) returns value of true, "
              << "where vc2 is the function object of s2."
              << endl;
    }
    else
    {
        cout << "vc2( 2,3 ) returns value of false, "
              << "where vc2 is the function object of s2."
              << endl;
    }
}

```

```

vc1( 2,3 ) returns value of true, where vc1 is the function object of s1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of s2.

```

set::value_compare

A type that provides a function object that can compare two element values to determine their relative order in the set.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the [set Class](#) topic.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for [value_comp](#) for an example of how to declare and use `value_compare`.

set::value_type

A type that describes an object stored as an element of a set in its capacity as a value.

```
typedef Key value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Key`.

For more information on `key`, see the Remarks section of the [set Class](#) topic.

Note that both [key_type](#) and `value_type` are synonyms for the template parameter `Key`. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    set <int> s1;
    set <int>::iterator s1_Iter;

    set <int>::value_type svt_Int;    // Declare value_type
    svt_Int = 10;                    // Initialize value_type

    set <int> :: key_type skt_Int;    // Declare key_type
    skt_Int = 20;                    // Initialize key_type

    s1.insert( svt_Int );             // Insert value into s1
    s1.insert( skt_Int );             // Insert key into s1

    // A set accepts key_types or value_types as elements
    cout << "The set has elements:";
    for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ) ; s1_Iter++)
        cout << " " << *s1_Iter;
    cout << "." << endl;
}
```

```
The set has elements: 10 20.
```

See also

[<set>](#)

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

multiset Class

2/19/2019 • 47 minutes to read • [Edit Online](#)

The C++ Standard Library multiset class is used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered. The key value of an element in a multiset may not be changed directly. Instead, old values must be deleted and elements with new values inserted.

Syntax

```
template <class Key, class Compare =less <Key>, class Allocator =allocator <Key>>
class multiset
```

Parameters

Key

The element data type to be stored in the multiset.

Compare

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multiset. The binary predicate **less**<Key> is the default value.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Allocator

The type that represents the stored allocator object that encapsulates details about the multiset's allocation and deallocation of memory. The default value is `allocator<Key>`.

Remarks

The C++ Standard Library multiset class is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple in the sense that its elements do not need to have unique keys, so that one key value can have many element values associated with it.
- A simple associative container because its element values are its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The iterator provided by the multiset class is a bidirectional iterator, but the class member functions [insert](#) and [multiset](#) have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts

form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`First` , `Last`) in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The multiset should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a multiset may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique key words, then a map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a multimap would be the container of choice.

The multiset orders the sequence it controls by calling a stored function object of type *Compare*. This stored object is a comparison function that may be accessed by calling the member function `key_comp`. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate $f(x, y)$ is a function object that has two argument objects x and y and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both $f(x, y)$ and $f(y, x)$ are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see [Heterogeneous Lookup in Associative Containers](#)

Constructors

CONSTRUCTOR	DESCRIPTION
<code>multiset</code>	Constructs a <code>multiset</code> that is empty or that is a copy of all or part of a specified <code>multiset</code> .

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A typedef for the <code>allocator</code> class for the <code>multiset</code> object.
<code>const_iterator</code>	A typedef for a bidirectional iterator that can read a const element in the <code>multiset</code> .
<code>const_pointer</code>	A typedef for a pointer to a const element in a <code>multiset</code> .

TYPE NAME	DESCRIPTION
<code>const_reference</code>	A typedef for a reference to a const element stored in a <code>multiset</code> for reading and performing const operations.
<code>const_reverse_iterator</code>	A typedef for a bidirectional iterator that can read any const element in the <code>multiset</code> .
<code>difference_type</code>	A signed integer typedef for the number of elements of a <code>multiset</code> in a range between elements pointed to by iterators.
<code>iterator</code>	A typedef for a bidirectional iterator that can read or modify any element in a <code>multiset</code> .
<code>key_compare</code>	A typedef for a function object that can compare two keys to determine the relative order of two elements in the <code>multiset</code> .
<code>key_type</code>	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the <code>multiset</code> .
<code>pointer</code>	A typedef for a pointer to an element in a <code>multiset</code> .
<code>reference</code>	A typedef for a reference to an element stored in a <code>multiset</code> .
<code>reverse_iterator</code>	A typedef for a bidirectional iterator that can read or modify an element in a reversed <code>multiset</code> .
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a <code>multiset</code> .
<code>value_compare</code>	The typedef for a function object that can compare two elements as sort keys to determine their relative order in the <code>multiset</code> .
<code>value_type</code>	A typedef that describes an object stored as an element as a <code>multiset</code> in its capacity as a value.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Returns an iterator that points to the first element in the <code>multiset</code> .
<code>cbegin</code>	Returns a const iterator that addresses the first element in the <code>multiset</code> .
<code>cend</code>	Returns a const iterator that addresses the location succeeding the last element in a <code>multiset</code> .

MEMBER FUNCTION	DESCRIPTION
<code>clear</code>	Erases all the elements of a <code>multiset</code> .
<code>count</code>	Returns the number of elements in a <code>multiset</code> whose key matches the key specified as a parameter.
<code>crbegin</code>	Returns a const iterator addressing the first element in a reversed set.
<code>crend</code>	Returns a const iterator that addresses the location succeeding the last element in a reversed set.
<code>emplace</code>	Inserts an element constructed in place into a <code>multiset</code> .
<code>emplace_hint</code>	Inserts an element constructed in place into a <code>multiset</code> , with a placement hint.
<code>empty</code>	Tests if a <code>multiset</code> is empty.
<code>end</code>	Returns an iterator that points to the location after the last element in a <code>multiset</code> .
<code>equal_range</code>	Returns a pair of iterators. The first iterator in the pair points to the first element in a <code>multiset</code> with a key that is greater than a specified key. The second iterator in the pair points to first element in the <code>multiset</code> with a key that is equal to or greater than the key.
<code>erase</code>	Removes an element or a range of elements in a <code>multiset</code> from specified positions or removes elements that match a specified key.
<code>find</code>	Returns an iterator that points to the first location of an element in a <code>multiset</code> that has a key equal to a specified key.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object that is used to construct the <code>multiset</code> .
<code>insert</code>	Inserts an element or a range of elements into a <code>multiset</code> .
<code>key_comp</code>	Provides a function object that can compare two sort keys to determine the relative order of two elements in the <code>multiset</code> .
<code>lower_bound</code>	Returns an iterator to the first element in a <code>multiset</code> with a key that is equal to or greater than a specified key.
<code>max_size</code>	Returns the maximum length of the <code>multiset</code> .
<code>rbegin</code>	Returns an iterator that points to the first element in a reversed <code>multiset</code> .

MEMBER FUNCTION	DESCRIPTION
<code>rend</code>	Returns an iterator that points to the location succeeding the last element in a reversed <code>multiset</code> .
<code>size</code>	Returns the number of elements in a <code>multiset</code> .
<code>swap</code>	Exchanges the elements of two <code>multiset</code> s.
<code>upper_bound</code>	Returns an iterator to the first element in a <code>multiset</code> with a key that is greater than a specified key.
<code>value_comp</code>	Retrieves a copy of the comparison object that is used to order element values in a <code>multiset</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Replaces the elements of a <code>multiset</code> with a copy of another <code>multiset</code> .

Requirements

Header: `<set>`

Namespace: `std`

`multiset::allocator_type`

A type that represents the allocator class for the `multiset` object

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the [multiset Class](#) topic.

Example

See the example for [get_allocator](#) for an example using `allocator_type`

`multiset::begin`

Returns an iterator addressing the first element in the `multiset`.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the `multiset` or the location succeeding an empty `multiset`.

Example

```
// multiset_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int>::iterator ms1_Iter;
    multiset <int>::const_iterator ms1_cIter;

    ms1.insert( 1 );
    ms1.insert( 2 );
    ms1.insert( 3 );

    ms1_Iter = ms1.begin( );
    cout << "The first element of ms1 is " << *ms1_Iter << endl;

    ms1_Iter = ms1.begin( );
    ms1.erase( ms1_Iter );

    // The following 2 lines would err as the iterator is const
    // ms1_cIter = ms1.begin( );
    // ms1.erase( ms1_cIter );

    ms1_cIter = ms1.begin( );
    cout << "The first element of ms1 is now " << *ms1_cIter << endl;
}
```

```
The first element of ms1 is 1
The first element of ms1 is now 2
```

multiset::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the [auto](#) type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

multiset::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

multiset::clear

Erases all the elements of a multiset.

```
void clear();
```

Example

```
// multiset_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;

    ms1.insert( 1 );
    ms1.insert( 2 );

    cout << "The size of the multiset is initially "
         << ms1.size( ) << "." << endl;

    ms1.clear( );
    cout << "The size of the multiset after clearing is "
         << ms1.size( ) << "." << endl;
}
```

```
The size of the multiset is initially 2.
The size of the multiset after clearing is 0.
```

multiset::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the multiset.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for [begin](#) for an example using `const_iterator`.

multiset::const_pointer

A type that provides a pointer to a **const** element in a multiset.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a multiset object.

multiset::const_reference

A type that provides a reference to a **const** element stored in a multiset for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// multiset_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;

    ms1.insert( 10 );
    ms1.insert( 20 );

    // Declare and initialize a const_reference &Ref1
    // to the 1st element
    const int &Ref1 = *ms1.begin( );

    cout << "The first element in the multiset is "
         << Ref1 << "." << endl;

    // The following line would cause an error because the
    // const_reference cannot be used to modify the multiset
    // Ref1 = Ref1 + 5;
}
```

The first element in the multiset is 10.

multiset::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the multiset.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the multiset in reverse.

Example

See the example for [rend](#) for an example of how to declare and use the `const_reverse_iterator`.

multiset::count

Returns the number of elements in a multiset whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the multiset.

Return Value

The number of elements in the multiset whose sort key matches the parameter key.

Remarks

The member function returns the number of elements x in the range

[lower_bound(*key*), upper_bound(*key*))

Example

The following example demonstrates the use of the `multiset::count` member function.

```
// multiset_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    multiset<int> ms1;
    multiset<int>::size_type i;

    ms1.insert(1);
    ms1.insert(1);
    ms1.insert(2);

    // Elements do not need to be unique in multiset,
    // so duplicates are allowed and counted.
    i = ms1.count(1);
    cout << "The number of elements in ms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = ms1.count(2);
    cout << "The number of elements in ms1 with a sort key of 2 is: "
         << i << "." << endl;

    i = ms1.count(3);
    cout << "The number of elements in ms1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

```
The number of elements in ms1 with a sort key of 1 is: 2.
The number of elements in ms1 with a sort key of 2 is: 1.
The number of elements in ms1 with a sort key of 3 is: 0.
```

multiset::crbegin

Returns a const iterator addressing the first element in a reversed multiset.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed multiset or addressing what had been the last element in the unreversed multiset.

Remarks

`crbegin` is used with a reversed multiset just as `begin` is used with a multiset.

With the return value of `crbegin`, the multiset object cannot be modified.

`crbegin` can be used to iterate through a multiset backwards.

Example

```
// multiset_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int>::const_reverse_iterator ms1_crIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_crIter = ms1.crbegin( );
    cout << "The first element in the reversed multiset is "
         << *ms1_crIter << "." << endl;
}
```

The first element in the reversed multiset is 30.

multiset::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed multiset.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multiset (the location that had preceded the first element in the unreversed multiset).

Remarks

`crend` is used with a reversed multiset just as `end` is used with a multiset.

With the return value of `crend`, the multiset object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its multiset.

The value returned by `crend` should not be dereferenced.

Example

```

// multiset_crend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
    using namespace std;
    multiset <int> ms1;
    multiset <int>::const_reverse_iterator ms1_crIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_crIter = ms1.crend( ) ;
    ms1_crIter--;
    cout << "The last element in the reversed multiset is "
         << *ms1_crIter << "." << endl;
}

```

multiset::difference_type

A signed integer type that can be used to represent the number of elements of a multiset in a range between elements pointed to by iterators.

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container.

The `difference_type` is typically used to represent the number of elements in the range `[first, last)` between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like `set`, subtraction between iterators is only supported by random-access iterators provided by a random-access container like `vector`.

Example


```

// multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
    using namespace std;

    multiset <int> ms1;
    multiset <int>::iterator ms1_iter, ms1_bIter, ms1_eIter;

    ms1.insert( 20 );
    ms1.insert( 10 );
    ms1.insert( 20 );

    ms1_bIter = ms1.begin( );
    ms1_eIter = ms1.end( );

    multiset <int>::difference_type  df_typ5, df_typ10, df_typ20;

    df_typ5 = count( ms1_bIter, ms1_eIter, 5 );
    df_typ10 = count( ms1_bIter, ms1_eIter, 10 );
    df_typ20 = count( ms1_bIter, ms1_eIter, 20 );

    // The keys, and hence the elements, of a multiset are not unique
    cout << "The number '5' occurs " << df_typ5
         << " times in multiset ms1.\n";
    cout << "The number '10' occurs " << df_typ10
         << " times in multiset ms1.\n";
    cout << "The number '20' occurs " << df_typ20
         << " times in multiset ms1.\n";

    // Count the number of elements in a multiset
    multiset <int>::difference_type  df_count = 0;
    ms1_iter = ms1.begin( );
    while ( ms1_iter != ms1_eIter)
    {
        df_count++;
        ms1_iter++;
    }

    cout << "The number of elements in the multiset ms1 is: "
         << df_count << "." << endl;
}

```

```

The number '5' occurs 0 times in multiset ms1.
The number '10' occurs 1 times in multiset ms1.
The number '20' occurs 2 times in multiset ms1.
The number of elements in the multiset ms1 is: 3.

```

multiset::emplace

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```

template <class... Args>
iterator emplace(Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the multiset.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state is not modified.

Example

```
// multiset_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    multiset<string> s1;

    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmin");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;

    s1.emplace("Bob");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;
}
```

multiset::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the multiset.
<i>where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>where</i> , insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state is not modified.

For a code example, see [set::emplace_hint](#).

multiset::empty

Tests if a multiset is empty.

```
bool empty() const;
```

Return Value

true if the multiset is empty; **false** if the multiset is nonempty.

Example

```
// multiset_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1, ms2;
    ms1.insert ( 1 );

    if ( ms1.empty( ) )
        cout << "The multiset ms1 is empty." << endl;
    else
        cout << "The multiset ms1 is not empty." << endl;

    if ( ms2.empty( ) )
        cout << "The multiset ms2 is empty." << endl;
    else
        cout << "The multiset ms2 is not empty." << endl;
}
```

```
The multiset ms1 is not empty.
The multiset ms2 is empty.
```

multiset::end

Returns the past-the-end iterator.

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the multiset is empty, then `multiset::end() == multiset::begin()`.

Remarks

end is used to test whether an iterator has passed the end of its multiset.

The value returned by **end** should not be dereferenced.

For a code example, see [multiset::find](#).

multiset::equal_range

Returns a pair of iterators respectively to the first element in a multiset with a key that is greater than a specified key and to the first element in the multiset with a key that is equal to or greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the multiset being searched.

Return Value

A pair of iterators such that the first is the [lower_bound](#) of the key and the second is the [upper_bound](#) of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first`, and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second`, and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```
// multiset_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    typedef multiset<int, less<int> > IntSet;
    IntSet ms1;
    multiset<int> :: const_iterator ms1_RcIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    pair<IntSet::const_iterator, IntSet::const_iterator> p1, p2;
    p1 = ms1.equal_range( 20 );

    cout << "The upper bound of the element with "
         << "a key of 20 in the multiset ms1 is: "
         << *( p1.second ) << "." << endl;

    cout << "The lower bound of the element with "
         << "a key of 20 in the multiset ms1 is: "
         << *( p1.first ) << "." << endl;

    // Compare the upper_bound called directly
    ms1_RcIter = ms1.upper_bound( 20 );
    cout << "A direct call of upper_bound( 20 ) gives "
         << *ms1_RcIter << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

    p2 = ms1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == ms1.end( ) ) && ( p2.second == ms1.end( ) ) )
        cout << "The multiset ms1 doesn't have an element "
             << "with a key less than 40." << endl;
    else
        cout << "The element of multiset ms1 with a key >= 40 is: "
             << *( p1.first ) << "." << endl;
}
```

The upper bound of the element with a key of 20 in the multiset ms1 is: 30.
The lower bound of the element with a key of 20 in the multiset ms1 is: 20.
A direct call of `upper_bound(20)` gives 30,
matching the 2nd element of the pair returned by `equal_range(20)`.
The multiset ms1 doesn't have an element with a key less than 40.

multiset::erase

Removes an element or a range of elements in a multiset from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the multiset if no such element exists.

For the third member function, returns the number of elements that have been removed from the multiset.

Remarks

For a code example, see [set::erase](#).

multiset::find

Returns an iterator that refers to the location of an element in a multiset that has a key equivalent to a specified key.

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key

The key value to be matched by the sort key of an element from the multiset being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the multiset (`multiset::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the multiset whose key is equivalent to the argument *key* under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the multiset object cannot be modified. If the return value of `find` is assigned to an `iterator`, the multiset object can be modified

Example

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    multiset<int> s1({ 40, 45 });
    cout << "The starting multiset s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate

    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified multiset s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

multiset::get_allocator

Returns a copy of the allocator object used to construct the multiset.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the multiset.

Remarks

Allocators for the multiset class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// multiset_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int>::allocator_type ms1_Alloc;
    multiset <int>::allocator_type ms2_Alloc;
    multiset <double>::allocator_type ms3_Alloc;
    multiset <int>::allocator_type ms4_Alloc;

    // The following lines declare objects
    // that use the default allocator.
    multiset <int> ms1;
    multiset <int, allocator<int> > ms2;
    multiset <double, allocator<double> > ms3;

    cout << "The number of integers that can be allocated"
         << endl << "before free memory is exhausted: "
         << ms2.max_size( ) << "." << endl;

    cout << "The number of doubles that can be allocated"
         << endl << "before free memory is exhausted: "
         << ms3.max_size( ) << "." << endl;

    // The following lines create a multiset ms4
    // with the allocator of multiset ms1
    ms1_Alloc = ms1.get_allocator( );
    multiset <int> ms4( less<int>( ), ms1_Alloc );
    ms4_Alloc = ms4.get_allocator( );

    // Two allocators are interchangeable if
    // storage allocated from each can be
    // deallocated with the other
    if( ms1_Alloc == ms4_Alloc )
    {
        cout << "Allocators are interchangeable."
             << endl;
    }
    else
    {
        cout << "Allocators are not interchangeable."
             << endl;
    }
}
```

multiset::insert

Inserts an element or a range of elements into a multiset.

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);
```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the multiset.
<i>Where</i>	The place to start searching for the correct point of insertion. (If that point immediately precedes <i>Where</i> , insertion can occur in amortized constant time instead of logarithmic time.)
<i>ValTy</i>	Template parameter that specifies the argument type that the multiset can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element

was inserted into the multiset.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the multiset.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The [value_type](#) of a container is a typedef that belongs to the container, and, for set, `multiset<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a multiset that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `s.insert(v.begin(), v.end());` inserts all elements of `v` into `s`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the multiset.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [multiset::emplace](#) and [multiset::emplace_hint](#).

Example

```
// multiset_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    // insert single values
    multiset<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    s1.insert(20);

    cout << "The original multiset values of s1 are:" << endl;
    print(s1);

    // intentionally attempt a duplicate, single element
    s1.insert(1);
    cout << "The modified multiset values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // single element, with hint
```

```

// single element, with hint
s1.insert(s1.end(), 30);
cout << "The modified multiset values of s1 are:" << endl;
print(s1);
cout << endl;

// The templated version inserting a jumbled range
multiset<int> s2;
vector<int> v;
v.push_back(43);
v.push_back(294);
v.push_back(41);
v.push_back(330);
v.push_back(42);
v.push_back(45);

cout << "Inserting the following vector data into s2:" << endl;
print(v);

s2.insert(v.begin(), v.end());

cout << "The modified multiset values of s2 are:" << endl;
print(s2);
cout << endl;

// The templated versions move-constructing elements
multiset<string> s3;
string str1("blue"), str2("green");

// single element
s3.insert(move(str1));
cout << "After the first move insertion, s3 contains:" << endl;
print(s3);

// single element with hint
s3.insert(s3.end(), move(str2));
cout << "After the second move insertion, s3 contains:" << endl;
print(s3);
cout << endl;

multiset<int> s4;
// Insert the elements from an initializer_list
s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
cout << "After initializer_list insertion, s4 contains:" << endl;
print(s4);
cout << endl;
}

```

multiset::iterator

A type that provides a constant [bidirectional iterator](#) that can read any element in a multiset.

```
typedef implementation-defined iterator;
```

Example

See the example for [begin](#) for an example of how to declare and use an `iterator`.

multiset::key_comp

Retrieves a copy of the comparison object used to order keys in a multiset.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the [multiset Class](#) topic.

Remarks

The stored object defines the member function:

`bool operator(const Key& x, const Key& y);`

which returns true if *x* strictly precedes *y* in the sort order.

Note that both [key_compare](#) and [value_compare](#) are synonyms for the template parameter `Compare`. Both types are provided for the classes `set` and `multiset`, where they are identical, for compatibility with the classes `map` and `multimap`, where they are distinct.

Example

```
// multiset_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;

    multiset <int, less<int> > ms1;
    multiset <int, less<int> >::key_compare kc1 = ms1.key_comp( ) ;
    bool result1 = kc1( 2, 3 ) ;
    if( result1 == true )
    {
        cout << "kc1( 2,3 ) returns value of true, "
              << "where kc1 is the function object of s1."
              << endl;
    }
    else
    {
        cout << "kc1( 2,3 ) returns value of false "
              << "where kc1 is the function object of ms1."
              << endl;
    }

    multiset <int, greater<int> > ms2;
    multiset <int, greater<int> >::key_compare kc2 = ms2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if( result2 == true )
    {
        cout << "kc2( 2,3 ) returns value of true, "
              << "where kc2 is the function object of ms2."
              << endl;
    }
    else
    {
        cout << "kc2( 2,3 ) returns value of false, "
              << "where kc2 is the function object of ms2."
              << endl;
    }
}
```

kc1(2,3) returns value of true, where kc1 is the function object of s1.
kc2(2,3) returns value of false, where kc2 is the function object of ms2.

multiset::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the multiset.

```
typedef Compare key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the [multiset Class](#) topic.

Example

See the example for [key_comp](#) for an example of how to declare and use `key_compare`.

multiset::key_type

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the multiset.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the [multiset Class](#) topic.

Example

See the example for [value_type](#) for an example of how to declare and use `key_type`.

multiset::lower_bound

Returns an iterator to the first element in a multiset with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(const Key& key) const;  
  
iterator lower_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the multiset being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a multiset that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the multiset if no match is found for the key.

Example

```

// multiset_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_RcIter = ms1.lower_bound( 20 );
    cout << "The element of multiset ms1 with a key of 20 is: "
         << *ms1_RcIter << "." << endl;

    ms1_RcIter = ms1.lower_bound( 40 );

    // If no match is found for the key, end( ) is returned
    if ( ms1_RcIter == ms1.end( ) )
        cout << "The multiset ms1 doesn't have an element "
             << "with a key of 40." << endl;
    else
        cout << "The element of multiset ms1 with a key of 40 is: "
             << *ms1_RcIter << "." << endl;

    // The element at a specific location in the multiset can be
    // found using a dereferenced iterator addressing the location
    ms1_AcIter = ms1.end( );
    ms1_AcIter--;
    ms1_RcIter = ms1.lower_bound( *ms1_AcIter );
    cout << "The element of ms1 with a key matching "
         << "that of the last element is: "
         << *ms1_RcIter << "." << endl;
}

```

The element of multiset ms1 with a key of 20 is: 20.
 The multiset ms1 doesn't have an element with a key of 40.
 The element of ms1 with a key matching that of the last element is: 30.

multiset::max_size

Returns the maximum length of the multiset.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the multiset.

Example

```
// multiset_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int>::size_type i;

    i = ms1.max_size( );
    cout << "The maximum possible length "
         << "of the multiset is " << i << "." << endl;
}
```

multiset::multiset

Constructs a multiset that is empty or that is a copy of all or part of some other multiset.

```

multiset();

explicit multiset (
    const Compare& Comp);

multiset (
    const Compare& Comp,
    const Allocator& Al);

multiset(
    const multiset& Right);

multiset(
    multiset&& Right);

multiset(
    initializer_list<Type> IList);

multiset(
    initializer_list<Type> IList,
    const Compare& Comp);

multiset(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last,
    const Compare& Comp);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last,
    const Compare& Comp,
    const Allocator& Al);

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The storage allocator class to be used for this multiset object, which defaults to <code>Allocator</code> .
<i>Comp</i>	The comparison function of type <code>const Compare</code> used to order the elements in the multiset, which defaults to <code>Compare</code> .
<i>Right</i>	The multiset of which the constructed multiset is to be a copy.
<i>First</i>	The position of the first element in the range of elements to be copied.

PARAMETER	DESCRIPTION
<i>Last</i>	The position of the first element beyond the range of elements to be copied.
<i>lList</i>	The initializer_list from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the multiset and that can later be returned by calling [get_allocator](#). The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their multiset.

All constructors store a function object of type Compare that is used to establish an order among the keys of the multiset and that can later be returned by calling [key_comp](#).

The first three constructors specify an empty initial multiset, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the multiset *Right*.

The fifth constructor specifies a copy of the multiset by moving *Right*.

The sixth, seventh, and eighth constructors specify an initializer_list from which to copy the elements.

The next three constructors copy the range [First, Last) of a multiset with increasing explicitness in specifying the type of comparison function and allocator.

Example

```
// multiset_ctor.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    //multiset <int>::iterator ms1_Iter, ms2_Iter, ms3_Iter;
    multiset <int>::iterator ms4_Iter, ms5_Iter, ms6_Iter, ms7_Iter;

    // Create an empty multiset ms0 of key type integer
    multiset <int> ms0;

    // Create an empty multiset ms1 with the key comparison
    // function of less than, then insert 4 elements
    multiset <int, less<int> > ms1;
    ms1.insert(10);
    ms1.insert(20);
    ms1.insert(20);
    ms1.insert(40);

    // Create an empty multiset ms2 with the key comparison
    // function of greater than, then insert 2 elements
    multiset <int, less<int> > ms2;
    ms2.insert(10);
    ms2.insert(20);

    // Create a multiset ms3 with the
    // allocator of multiset ms1
    multiset <int>::allocator_type ms1_Alloc;
```

```

ms1_Alloc = ms1.get_allocator();
multiset<int> ms3(less<int>(), ms1_Alloc);
ms3.insert(30);

// Create a copy, multiset ms4, of multiset ms1
multiset<int> ms4(ms1);

// Create a multiset ms5 by copying the range ms1[ first, last)
multiset<int>::const_iterator ms1_bcIter, ms1_ecIter;
ms1_bcIter = ms1.begin();
ms1_ecIter = ms1.begin();
ms1_ecIter++;
ms1_ecIter++;
multiset<int> ms5(ms1_bcIter, ms1_ecIter);

// Create a multiset ms6 by copying the range ms4[ first, last)
// and with the allocator of multiset ms2
multiset<int>::allocator_type ms2_Alloc;
ms2_Alloc = ms2.get_allocator();
multiset<int> ms6(ms4.begin(), ++ms4.begin(), less<int>(), ms2_Alloc);

cout << "ms1 =";
for (auto i : ms1)
    cout << " " << i;
cout << endl;

cout << "ms2 =";
for (auto i : ms2)
    cout << " " << i;
cout << endl;

cout << "ms3 =";
for (auto i : ms3)
    cout << " " << i;
cout << endl;

cout << "ms4 =";
for (auto i : ms4)
    cout << " " << i;
cout << endl;

cout << "ms5 =";
for (auto i : ms5)
    cout << " " << i;
cout << endl;

cout << "ms6 =";
for (auto i : ms6)
    cout << " " << i;
cout << endl;

// Create a multiset by moving ms5
multiset<int> ms7(move(ms5));
cout << "ms7 =";
for (auto i : ms7)
    cout << " " << i;
cout << endl;

// Create a multiset with an initializer_list
multiset<int> ms8({1, 2, 3, 4});
cout << "ms8=";
for (auto i : ms8)
    cout << " " << i;
cout << endl;
}

```

multiset::operator=

Replaces the elements of this `multiset` using elements from another `multiset`.

```
multiset& operator=(const multiset& right);

multiset& operator=(multiset&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>multiset</code> from which elements are copied or moved.

Remarks

`operator=` copies or moves the elements in *right* into this `multiset`, depending on the reference type (lvalue or rvalue) used. Elements that are in this `multiset` before `operator=` executes are discarded.

Example

```
// multiset_operator_as.cpp
// compile with: /EHsc
#include <multiset>
#include <iostream>

int main( )
{
    using namespace std;
    multiset<int> v1, v2, v3;
    multiset<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

multiset::pointer

A type that provides a pointer to an element in a multiset.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an [iterator](#) should be used to access the elements in a multiset object.

multiset::rbegin

Returns an iterator addressing the first element in a reversed multiset.

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed multiset or addressing what had been the last element in the unreversed multiset.

Remarks

`rbegin` is used with a reversed multiset just as `rbegin` is used with a multiset.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the multiset object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the multiset object can be modified.

`rbegin` can be used to iterate through a multiset backwards.

Example

```

// multiset_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int>::iterator ms1_iter;
    multiset <int>::reverse_iterator ms1_rIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_rIter = ms1.rbegin( );
    cout << "The first element in the reversed multiset is "
         << *ms1_rIter << "." << endl;

    // begin can be used to start an iteration
    // through a multiset in a forward order
    cout << "The multiset is:";
    for ( ms1_iter = ms1.begin( ) ; ms1_iter != ms1.end( ); ms1_iter++ )
        cout << " " << *ms1_iter;
    cout << endl;

    // rbegin can be used to start an iteration
    // through a multiset in a reverse order
    cout << "The reversed multiset is:";
    for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
        cout << " " << *ms1_rIter;
    cout << endl;

    // A multiset element can be erased by dereferencing to its key
    ms1_rIter = ms1.rbegin( );
    ms1.erase ( *ms1_rIter );

    ms1_rIter = ms1.rbegin( );
    cout << "After the erasure, the first element "
         << "in the reversed multiset is "<< *ms1_rIter << "."
         << endl;
}

```

```

The first element in the reversed multiset is 30.
The multiset is: 10 20 30
The reversed multiset is: 30 20 10
After the erasure, the first element in the reversed multiset is 20.

```

multiset::reference

A type that provides a reference to an element stored in a multiset.

```

typedef typename allocator_type::reference reference;

```

Example

```
// multiset_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;

    ms1.insert( 10 );
    ms1.insert( 20 );

    // Declare and initialize a reference &Ref1 to the 1st element
    const int &Ref1 = *ms1.begin( );

    cout << "The first element in the multiset is "
         << Ref1 << "." << endl;
}
```

```
The first element in the multiset is 10.
```

multiset::rend

Returns an iterator that addresses the location succeeding the last element in a reversed multiset.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multiset (the location that had preceded the first element in the unreversed multiset).

Remarks

`rend` is used with a reversed multiset just as `end` is used with a multiset.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the multiset object cannot be modified.

If the return value of `rend` is assigned to a `reverse_iterator`, then the multiset object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its multiset.

The value returned by `rend` should not be dereferenced.

Example

```

// multiset_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
    using namespace std;
    multiset <int> ms1;
    multiset <int>::iterator ms1_Iter;
    multiset <int>::reverse_iterator ms1_rIter;
    multiset <int>::const_reverse_iterator ms1_crIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_rIter = ms1.rend( ) ;
    ms1_rIter--;
    cout << "The last element in the reversed multiset is "
         << *ms1_rIter << "." << endl;

    // end can be used to terminate an iteration
    // through a multiset in a forward order
    cout << "The multiset is: ";
    for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
        cout << *ms1_Iter << " ";
    cout << "." << endl;

    // rend can be used to terminate an iteration
    // through a multiset in a reverse order
    cout << "The reversed multiset is: ";
    for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
        cout << *ms1_rIter << " ";
    cout << "." << endl;

    ms1_rIter = ms1.rend( );
    ms1_rIter--;
    ms1.erase ( *ms1_rIter );

    ms1_rIter = ms1.rend( );
    --ms1_rIter;
    cout << "After the erasure, the last element in the "
         << "reversed multiset is " << *ms1_rIter << "." << endl;
}

```

multiset::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed multiset.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the multiset in reverse.

Example

See example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

multiset::size

Returns the number of elements in the multiset.

```
size_type size() const;
```

Return Value

The current length of the multiset.

Example

```
// multiset_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int> :: size_type i;

    ms1.insert( 1 );
    i = ms1.size( );
    cout << "The multiset length is " << i << "." << endl;

    ms1.insert( 2 );
    i = ms1.size( );
    cout << "The multiset length is now " << i << "." << endl;
}
```

```
The multiset length is 1.
The multiset length is now 2.
```

multiset::size_type

An unsigned integer type that can represent the number of elements in a multiset.

```
typedef typename allocator_type::size_type size_type;
```

Example

See example for [size](#) for an example of how to declare and use `size_type`

multiset::swap

Exchanges the elements of two multisets.

```
void swap(
    multiset<Key, Compare, Allocator>& right);
```

Parameters

right

The argument multiset providing the elements to be swapped with the target multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two multisets whose elements are being exchanged.

Example

```
// multiset_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1, ms2, ms3;
    multiset <int>::iterator ms1_Iter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );
    ms2.insert( 100 );
    ms2.insert( 200 );
    ms3.insert( 300 );

    cout << "The original multiset ms1 is:";
    for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
        cout << " " << *ms1_Iter;
    cout << "." << endl;

    // This is the member function version of swap
    ms1.swap( ms2 );

    cout << "After swapping with ms2, list ms1 is:";
    for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
        cout << " " << *ms1_Iter;
    cout << "." << endl;

    // This is the specialized template version of swap
    swap( ms1, ms3 );

    cout << "After swapping with ms3, list ms1 is:";
    for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
        cout << " " << *ms1_Iter;
    cout << "." << endl;
}
```

```
The original multiset ms1 is: 10 20 30.
After swapping with ms2, list ms1 is: 100 200.
After swapping with ms3, list ms1 is: 300.
```

multiset::upper_bound

Returns an iterator to the first element in a multiset with a key that is greater than a specified key.

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

key

The argument *key* to be compared with the sort key of an element from the multiset being searched.

Return Value

An **iterator** or `const_iterator` that addresses the location of an element in a multiset with a key that is greater

than the argument key, or that addresses the location succeeding the last element in the multiset if no match is found for the key.

Example

```
// multiset_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_RcIter = ms1.upper_bound( 20 );
    cout << "The first element of multiset ms1 with a key greater "
         << "than 20 is: " << *ms1_RcIter << "." << endl;

    ms1_RcIter = ms1.upper_bound( 30 );

    // If no match is found for the key, end( ) is returned
    if ( ms1_RcIter == ms1.end( ) )
        cout << "The multiset ms1 doesn't have an element "
             << "with a key greater than 30." << endl;
    else
        cout << "The element of multiset ms1 with a key > 40 is: "
             << *ms1_RcIter << "." << endl;

    // The element at a specific location in the multiset can be
    // found using a dereferenced iterator addressing the location
    ms1_AcIter = ms1.begin( );
    ms1_RcIter = ms1.upper_bound( *ms1_AcIter );
    cout << "The first element of ms1 with a key greater than"
         << endl << "that of the initial element of ms1 is: "
         << *ms1_RcIter << "." << endl;
}
```

The first element of multiset ms1 with a key greater than 20 is: 30.
The multiset ms1 doesn't have an element with a key greater than 30.
The first element of ms1 with a key greater than
that of the initial element of ms1 is: 20.

multiset::value_comp

Retrieves a copy of the comparison object used to order element values in a multiset.

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the [multiset Class](#) topic.

Remarks

The stored object defines the member function:

```
bool operator( const Key& _xVal, const Key& _yVal );
```

which returns true if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter `Compare`. Both types are provided for the classes `set` and `multiset`, where they are identical, for compatibility with the classes `map` and `multimap`, where they are distinct.

Example

```
// multiset_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;

    multiset <int, less<int> > ms1;
    multiset <int, less<int> >::value_compare vc1 = ms1.value_comp( );
    bool result1 = vc1( 2, 3 );
    if( result1 == true )
    {
        cout << "vc1( 2,3 ) returns value of true, "
              << "where vc1 is the function object of ms1."
              << endl;
    }
    else
    {
        cout << "vc1( 2,3 ) returns value of false, "
              << "where vc1 is the function object of ms1."
              << endl;
    }

    set <int, greater<int> > ms2;
    set<int, greater<int> >::value_compare vc2 = ms2.value_comp( );
    bool result2 = vc2( 2, 3 );
    if( result2 == true )
    {
        cout << "vc2( 2,3 ) returns value of true, "
              << "where vc2 is the function object of ms2."
              << endl;
    }
    else
    {
        cout << "vc2( 2,3 ) returns value of false, "
              << "where vc2 is the function object of ms2."
              << endl;
    }
}
```

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of ms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of ms2.
```

multiset::value_compare

The type that provides a function object that can compare two sort keys to determine their relative order in the `multiset`.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Compare`.

Note that both `key_compare` and `value_compare` are synonyms for the template parameter `Compare`. Both types are provided for the classes `set` and `multiset`, where they are identical, for compatibility with the classes `map` and `multimap`, where they are distinct.

For more information on `Compare`, see the Remarks section of the [multiset Class](#) topic.

Example

See the example for `value_comp` for an example of how to declare and use `value_compare`.

multiset::value_type

A type that describes an object stored as an element as a multiset in its capacity as a value.

```
typedef Key value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Key`.

Note that both `key_type` and `value_type` are synonyms for the template parameter `Key`. Both types are provided for the classes `set` and `multiset`, where they are identical, for compatibility with the classes `map` and `multimap`, where they are distinct.

For more information on `Key`, see the Remarks section of the topic.

Example

```
// multiset_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
    using namespace std;
    multiset <int> ms1;
    multiset <int>::iterator ms1_Iter;

    multiset <int> :: value_type svt_Int; // Declare value_type
    svt_Int = 10;                        // Initialize value_type

    multiset <int> :: key_type skt_Int; // Declare key_type
    skt_Int = 20;                      // Initialize key_type

    ms1.insert( svt_Int );              // Insert value into s1
    ms1.insert( skt_Int );              // Insert key into s1

    // A multiset accepts key_types or value_types as elements
    cout << "The multiset has elements:";
    for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ) ; ms1_Iter++ )
        cout << " " << *ms1_Iter;
    cout << "." << endl;
}
```

```
The multiset has elements: 10 20.
```

See also

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<shared_mutex>

3/28/2019 • 4 minutes to read • [Edit Online](#)

The <shared_mutex> header provides synchronization primitives for protection of shared data that can be accessed by multiple threads. In addition to the exclusive access control provided by mutex classes, the shared mutex classes also allow shared ownership by multiple threads for non-exclusive access. Shared mutexes can be used to control resources that can be read by several threads without causing a race condition, but must be written exclusively by a single thread.

The header <shared_mutex> defines the classes `shared_mutex` and `shared_timed_mutex`, the template class `shared_lock`, and the template function `swap` for shared mutex support.

CLASSES	DESCRIPTION
shared_mutex Class	A shared mutex type that can be locked exclusively by one agent or shared non-exclusively by multiple agents.
shared_timed_mutex Class	A shared timed mutex type that can be locked exclusively by one agent or shared non-exclusively by multiple agents.
shared_lock Class	A template class that wraps a shared mutex to support timed lock operations and non-exclusive sharing by multiple agents.
FUNCTIONS	DESCRIPTION
swap	Swaps the content of the shared mutex objects referenced by the function parameters.

Syntax

```
namespace std {
    class shared_mutex;
    class shared_timed_mutex;
    template <class Mutex>
    class shared_lock;
    template <class Mutex>
    void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}
```

Remarks

An instance of the class `shared_mutex` is a *shared mutex type*, a type that controls the shared ownership of a mutex within a scope. A shared mutex type meets all the requirements of a mutex type, as well as members to support shared non-exclusive ownership.

A shared mutex type supports the additional methods `lock_shared`, `unlock_shared`, and `try_lock_shared`:

- The `lock_shared` method blocks the calling thread until the thread obtains shared ownership of the mutex.
- The `unlock_shared` method releases shared ownership of the mutex held by the calling thread.

- The `try_lock_shared` method tries to obtain shared ownership of the mutex without blocking. Its return type is convertible to **bool** and is **true** if the method obtains ownership, but is otherwise **false**.

The class `shared_timed_mutex` is a *shared timed mutex type*, a type that meets the requirements of both a shared mutex type and a timed mutex type.

A shared timed mutex type supports the additional methods `try_lock_shared_for` and `try_lock_shared_until`:

- The `try_lock_shared_for` method attempts to obtain shared ownership of the mutex until the duration specified by the parameter has passed. If the duration is not positive, the method is equivalent to `try_lock_shared`. The method does not return within the duration specified unless shared ownership is obtained. Its return value is **true** if the method obtains ownership, but is otherwise **false**.
- The `try_lock_shared_until` method attempts to obtain shared ownership of the mutex until the specified absolute time has passed. If the specified time has already passed, the method is equivalent to `try_lock_shared`. The method does not return before the time specified unless shared ownership is obtained. Its return value is **true** if the method obtains ownership, but is otherwise **false**.

The `shared_lock` template class extends support for timed locking and transfer of ownership to a shared mutex. Ownership of the mutex may be obtained at or after construction, and may be transferred to another `shared_lock` object. Objects of type `shared_lock` can be moved, but not copied.

WARNING

Beginning in Visual Studio 2015, the C++ Standard Library synchronization types are based on Windows synchronization primitives and no longer use ConCRT (except when the target platform is Windows XP). The types defined in `<shared_mutex>` should not be used with any ConCRT types or functions.

Classes

shared_mutex Class

Class `shared_mutex` implements a non-recursive mutex with shared ownership semantics.

```
class shared_mutex {
public:
    shared_mutex();
    ~shared_mutex();
    shared_mutex(const shared_mutex&) = delete;
    shared_mutex& operator=(const shared_mutex&) = delete;
    // Exclusive ownership
    void lock();
    // blocking
    bool try_lock();
    void unlock();
    // Shared ownership
    void lock_shared();
    // blocking
    bool try_lock_shared();
    void unlock_shared();
    // Getters
    typedef void** native_handle_type; // implementation defined
    native_handle_type native_handle();
};
```

shared_timed_mutex Class

Class `shared_timed_mutex` implements a non-recursive mutex with shared ownership semantics that meets the requirements of a timed mutex type.

```

class shared_timed_mutex {
public:
    shared_timed_mutex();
    ~shared_timed_mutex();
    shared_timed_mutex(const shared_timed_mutex&) = delete;
    shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
    // Exclusive ownership
    void lock();
    // blocking
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();
    // Shared ownership
    void lock_shared();
    // blocking
    bool try_lock_shared();
    template <class Rep, class Period>
    bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock_shared();
};

```

shared_lock Class

Template class `shared_lock` controls the shared ownership of a shared mutex object within a scope. The template parameter must be a shared mutex type.


```

class shared_lock {
public:
    typedef Mutex mutex_type;
    shared_lock() noexcept;
    explicit shared_lock(mutex_type& m);
    // blocking
    shared_lock(mutex_type& m, defer_lock_t) noexcept;
    shared_lock(mutex_type& m, try_to_lock_t);
    shared_lock(mutex_type& m, adopt_lock_t);
    template <class Clock, class Duration>
    shared_lock(mutex_type& m,
        const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
    shared_lock(mutex_type& m,
        const chrono::duration<Rep, Period>& rel_time);
    ~shared_lock();
    shared_lock(shared_lock const&) = delete;
    shared_lock& operator=(shared_lock const&) = delete;
    shared_lock(shared_lock&& u) noexcept;
    shared_lock& operator=(shared_lock&& u) noexcept;
    void lock();
    // blocking
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();
    // Setters
    void swap(shared_lock& u) noexcept;
    mutex_type* release() noexcept;
    // Getters
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;
private:
    mutex_type* pm; // exposition only
    bool owns; // exposition only
};

```

Functions

swap

Swaps the `shared_lock` objects.

```

template <class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;

```

Exchanges the content of two `shared_lock` objects. Effectively the same as `x.swap(y)`.

Requirements

Header: `<shared_mutex>`

Namespace: `std`

See also

[Header Files Reference](#)

[<mutex>](#)

<sstream>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class [basic_string](#).

Syntax

```
namespace std {
    template <class CharType, class Traits = char_traits<CharType>, class Allocator = allocator<CharType>>
    class basic_stringbuf;
    typedef basic_stringbuf<char>
    stringbuf;
    typedef basic_stringbuf<wchar_t> wstringbuf;

    template <class CharType, class Traits = char_traits<CharType>, class Allocator = allocator<CharType>>
    class basic_istream;
    typedef basic_istream<char>
    istream;
    typedef basic_istream<wchar_t> wistream;

    template <class CharType, class Traits = char_traits<CharType>, class Allocator = allocator<CharType>>
    class basic_ostream;
    typedef basic_ostream<char>
    ostream;
    typedef basic_ostream<wchar_t> wostream;

    template <class CharType, class Traits = char_traits<CharType>, class Allocator = allocator<CharType>>
    class basic_stringstream;
    typedef basic_stringstream<char>
    stringstream;
    typedef basic_stringstream<wchar_t> wstringstream;
    // TEMPLATE FUNCTIONS
    template <class CharType, class Traits, class Allocator>
    void swap(
        basic_stringbuf<CharType, Traits, Allocator>& left,
        basic_stringbuf<CharType, Traits, Allocator>& right);

    template <class CharType, class Traits, class Allocator>
    void swap(
        basic_istream<CharType, Traits, Allocator>& left,
        basic_istream<CharType, Traits, Allocator>& right);

    template <class CharType, class Traits, class Allocator>
    void swap(
        basic_ostream<CharType, Traits, Allocator>& left,
        basic_ostream<CharType, Traits, Allocator>& right);

    template <class CharType, class Traits, class Allocator>
    void swap (
        basic_stringstream<CharType, Traits, Allocator>& left,
        basic_stringstream<CharType, Traits, Allocator>& right);

} // namespace std
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	Reference to an <code>sstream</code> object.
<i>right</i>	Reference to an <code>sstream</code> object.

Remarks

Objects of type `char *` can use the functionality in [<sstream>](#) for streaming. However, [<sstream>](#) is deprecated and the use of [<stream>](#) is encouraged.

Typedefs

TYPE NAME	DESCRIPTION
istringstream	Creates a type <code>basic_istringstream</code> specialized on a char template parameter.
ostringstream	Creates a type <code>basic_ostringstream</code> specialized on a char template parameter.
stringbuf	Creates a type <code>basic_stringbuf</code> specialized on a char template parameter.
stringstream	Creates a type <code>basic_stringstream</code> specialized on a char template parameter.
wistringstream	Creates a type <code>basic_istringstream</code> specialized on a wchar_t template parameter.
wostringstream	Creates a type <code>basic_ostringstream</code> specialized on a wchar_t template parameter.
wstringbuf	Creates a type <code>basic_stringbuf</code> specialized on a wchar_t template parameter.
wstringstream	Creates a type <code>basic_stringstream</code> specialized on a wchar_t template parameter.

Manipulators

swap	Exchanges the values between two <code>sstream</code> objects.
----------------------	--

Classes

CLASS	DESCRIPTION
basic_stringbuf	Describes a stream buffer that controls the transmission of elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> , to and from a sequence of elements stored in an array object.

CLASS	DESCRIPTION
basic_istream	Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf < Elem , Tr , <code>Alloc</code> >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> , and whose elements are allocated by an allocator of class <code>Alloc</code> .
basic_ostringstream	Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_stringbuf < Elem , Tr , <code>Alloc</code> >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> , and whose elements are allocated by an allocator of class <code>Alloc</code> .
basic_stringstream	Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf < Elem , Tr , <code>Alloc</code> >, with elements of type <code>Elem</code> , whose character traits are determined by the class <code>Tr</code> , and whose elements are allocated by an allocator of class <code>Alloc</code> .

Requirements

- **Header:** `<sstream>`
- **Namespace:** `std`

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<sstream> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

swap

Exchanges the values between two `sstream` objects.

```
template <class Elem, class Tr, class Alloc>
void swap(
    basic_stringbuf<Elem, Tr, Alloc>& left,
    basic_stringbuf<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_istringstream<Elem, Tr, Alloc>& left,
    basic_istringstream<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_ostringstream<Elem, Tr, Alloc>& left,
    basic_ostringstream<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_stringstream<Elem, Tr, Alloc>& left,
    basic_stringstream<Elem, Tr, Alloc>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	Reference to an <code>sstream</code> object.
<i>right</i>	Reference to an <code>sstream</code> object.

Remarks

The template function executes `left.swap(right)`.

See also

[<sstream>](#)

<sstream> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

istringstream	ostreamstream	stringbuf
stringstream	wstringstream	wostringstream
wstringbuf	wstringstream	

istringstream

Creates a type `basic_istringstream` specialized on a **char** template parameter.

```
typedef basic_istringstream<char> istringstream;
```

Remarks

The type is a synonym for template class [basic_istringstream](#), specialized for elements of type **char**.

ostreamstream

Creates a type `basic_ostreamstream` specialized on a **char** template parameter.

```
typedef basic_ostreamstream<char> ostreamstream;
```

Remarks

The type is a synonym for template class [basic_ostreamstream](#), specialized for elements of type **char**.

stringbuf

Creates a type `basic_stringbuf` specialized on a **char** template parameter.

```
typedef basic_stringbuf<char> stringbuf;
```

Remarks

The type is a synonym for template class [basic_stringbuf](#), specialized for elements of type **char**.

stringstream

Creates a type `basic_stringstream` specialized on a **char** template parameter.

```
typedef basic_stringstream<char> stringstream;
```

Remarks

The type is a synonym for template class [basic_stringstream](#), specialized for elements of type **char**.

wstringstream

Creates a type `basic_istringstream` specialized on a **wchar_t** template parameter.

```
typedef basic_istringstream<wchar_t> wstringstream;
```

Remarks

The type is a synonym for template class [basic_istringstream](#), specialized for elements of type **wchar_t**.

wostringstream

Creates a type `basic_ostringstream` specialized on a **wchar_t** template parameter.

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

Remarks

The type is a synonym for template class [basic_ostringstream](#), specialized for elements of type **wchar_t**.

wstringbuf

Creates a type `basic_stringbuf` specialized on a **wchar_t** template parameter.

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

Remarks

The type is a synonym for template class [basic_stringbuf](#), specialized for elements of type **wchar_t**.

wstringstream

Creates a type `basic_stringstream` specialized on a **wchar_t** template parameter.

```
typedef basic_stringstream<wchar_t> wstringstream;
```

Remarks

The type is a synonym for template class [basic_stringstream](#), specialized for elements of type **wchar_t**.

See also

[<sstream>](#)

basic_stringbuf Class

11/8/2018 • 9 minutes to read • [Edit Online](#)

Describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are determined by the class `Tr`, to and from a sequence of elements stored in an array object.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>,  
         class Alloc = allocator<Elem>>  
class basic_stringbuf : public basic_streambuf<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `basic_stringbuf<Elem, Tr, Alloc>` stores a copy of the `ios_base::openmode` argument from its constructor as its `stringbuf` mode **mode**:

- If `mode & ios_base::in` is nonzero, the input buffer is accessible. For more information, see [basic_streambuf Class](#).
- If `mode & ios_base::out` is nonzero, the output buffer is accessible.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_stringbuf	Constructs an object of type <code>basic_stringbuf</code> .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	The type is a synonym for the template parameter <i>Alloc</i> .
char_type	Associates a type name with the <i>Elem</i> template parameter.
int_type	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <i>Tr</i> scope.

TYPE NAME	DESCRIPTION
<code>off_type</code>	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <i>Tr</i> scope.
<code>pos_type</code>	Makes this type within <code>basic_filebuf</code> 's scope equivalent to the type of the same name in the <i>Tr</i> scope.
<code>traits_type</code>	Associates a type name with the <i>Tr</i> template parameter.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>overflow</code>	A protected, virtual function that can be called when a new character is inserted into a full buffer.
<code>pbackfail</code>	The protected virtual member function tries to put back an element into the input buffer, then makes it the current element (pointed to by the next pointer).
<code>seekoff</code>	The protected virtual member function tries to alter the current positions for the controlled streams.
<code>seekpos</code>	The protected virtual member function tries to alter the current positions for the controlled streams.
<code>str</code>	Sets or gets the text in a string buffer without changing the write position.
<code>swap</code>	
<code>underflow</code>	The protected virtual member function to extract the current element from the input stream.

Requirements

Header: `<sstream>`

Namespace: `std`

`basic_stringbuf::allocator_type`

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

`basic_stringbuf::basic_stringbuf`

Constructs an object of type `basic_stringbuf`.

```

basic_stringbuf(
    ios_base::openmode _Mode = ios_base::in | ios_base::out);

basic_stringbuf(
    const basic_string<Elem, Tr, Alloc>& str,
    ios_base::openmode _Mode = ios_base::in | ios_base::out);

```

Parameters

_Mode

One of the enumerations in [ios_base::openmode](#).

str

An object of type [basic_string](#).

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. For more information, see the Remarks section of the [basic_streambuf Class](#). It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the [basic_stringbuf Class](#).

The second constructor allocates a copy of the sequence controlled by the string object *str*. If

`_Mode & ios_base::in` is nonzero, it sets the input buffer to start reading at the start of the sequence. If

`_Mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the [basic_stringbuf Class](#).

basic_stringbuf::char_type

Associates a type name with the *Elem* template parameter.

```
typedef Elem char_type;
```

basic_stringbuf::int_type

Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::int_type int_type;
```

basic_stringbuf::off_type

Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::off_type off_type;
```

basic_stringbuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int_type overflow(int_type _Meta = traits_type::eof());
```

Parameters

_Meta

The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof`. Otherwise, it returns **traits_type::not_eof**(*_Meta*).

Remarks

If *_Meta* does not compare equal to **traits_type::eof**, the protected virtual member function tries to insert the element **traits_type::to_char_type**(*_Meta*) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. Extending the output buffer this way also extends any associated input buffer.

basic_stringbuf::pbackfail

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(int_type _Meta = traits_type::eof());
```

Parameters

_Meta

The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof`. Otherwise, it returns **traits_type::not_eof**(*_Meta*).

Remarks

If *_Meta* compares equal to **traits_type::eof**, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **byte** = **traits_type::to_char_type**(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to byte, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode permits the sequence to be altered (**mode & ios_base::out** is nonzero), it can store byte into the putback position and decrement the next pointer for the input buffer.

basic_stringbuf::pos_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the `Tr` scope.

```
typedef typename traits_type::pos_type pos_type;
```

basic_stringbuf::seekoff

The protected virtual member function tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(
    off_type _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

_Off

The position to seek for relative to *_Way*. For more information, see [basic_stringbuf::off_type](#).

_Way

The starting point for offset operations. See [ios_base::seekdir](#) for possible values.

_Mode

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions. For more information, see [ios_base::openmode](#).

Return Value

Returns the new position or an invalid stream position.

Remarks

For an object of class `basic_stringbuf<Elem, Tr, Alloc>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `_Way == ios_base::beg`, the new position is the beginning of the stream plus *_Off*.
- If `_Way == ios_base::cur`, the new position is the current stream position plus *_Off*.
- If `_Way == ios_base::end`, the new position is the end of the stream plus *_Off*.

If `_Mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `_Mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, *_Way* must be `ios_base::beg` or `ios_base::end` and both streams are positioned at the same element. Otherwise (or if neither position is affected), the positioning operation fails.

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::seekpos

The protected virtual member function tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(pos_type _Sp, ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

_Sp

The position to seek for.

_Mode

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

For an object of class `basic_stringbuf< Elem, Tr, Alloc >`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `_Sp`.

If **mode & ios_base::in** is nonzero, the function alters the next position to read in the input buffer. If **mode & ios_base::out** is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected), the positioning operation fails.

basic_stringbuf::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str() const;
void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

`_Newstr`

The new string.

Return Value

Returns an object of class `basic_string< Elem, Tr, Alloc >`, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns an object of class `basic_string< Elem, Tr, Alloc >`, whose controlled sequence is a copy of the sequence controlled by ***this**. The sequence copied depends on the stored stringbuf mode:

- If **mode & ios_base::out** is nonzero and an output buffer exists, the sequence is the entire output buffer (`egptr` - `pbase` elements beginning with `pbase`).
- If **mode & ios_base::in** is nonzero and an input buffer exists, the sequence is the entire input buffer (`egptr` - `eback` elements beginning with `eback`).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by ***this**. It then allocates a copy of the sequence controlled by `_Newstr`. If **mode & ios_base::in** is nonzero, it sets the input buffer to start reading at the beginning of the sequence. If **mode & ios_base::out** is nonzero, it sets the output buffer to start writing at the beginning of the sequence.

Example

```
// basic_stringbuf_str.cpp
// compile with: /EHsc
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
    basic_string<char> i( "test" );
    stringstream ss;

    ss.rdbuf( )->str( i );
    cout << ss.str( ) << endl;

    ss << "z";
    cout << ss.str( ) << endl;

    ss.rdbuf( )->str( "be" );
    cout << ss.str( ) << endl;
}
```

```
test
zest
be
```

basic_stringbuf::traits_type

Associates a type name with the *Tr* template parameter.

```
typedef Tr traits_type;
```

Remarks

The type is a synonym for the template parameter *Tr*.

basic_stringbuf::underflow

Protected, virtual function to extract the current element from the input stream.

```
virtual int_type underflow();
```

Return Value

If the function cannot succeed, it returns **traits_type::eof**. Otherwise, it returns the current element in the input stream, which are converted.

Remarks

The protected virtual member function tries to extract the current element `byte` from the input buffer, advance the current stream position, and return the element as **traits_type::to_int_type(byte)**. It can do so in one way: If a read position is available, it takes `byte` as the element stored in the read position and advances the next pointer for the input buffer.

basic_streambuf::swap

Swaps the contents of this string buffer with another string buffer.

```
void basic_stringbuf<T>::swap(basic_stringbuf& other)
```

Parameters

other

The `basic_stringbuf` whose contents will be swapped with this `basic_stringbuf`.

Remarks

`basic_stringbuf::operator=`

Assigns the contents of the `basic_stringbuf` on the right side of the operator to the `basic_stringbuf` on the left side.

```
basic_stringbuf& basic_stringbuf:: operator=(const basic_stringbuf& other)
```

Parameters

other

A `basic_stringbuf` whose contents, including locale traits, will be assigned to the `stringbuf` on the left side of the operator.

Remarks

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_istream Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_istream : public basic_istream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >, with elements of type *Elem*, whose character traits are determined by the class *Tr*, and whose elements are allocated by an allocator of class *Alloc*. The object stores an object of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_istream	Constructs an object of type <code>basic_istream</code> .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	The type is a synonym for the template parameter <code>Alloc</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
rdbuf	Returns the address of the stored stream buffer of type <code>pointer</code> to basic_stringbuf < <code>Elem</code> , <code>Tr</code> , <code>Alloc</code> >.
str	Sets or gets the text in a string buffer without changing the write position.

MEMBER FUNCTION	DESCRIPTION
swap	Exchanges the values in this <code>basic_istream</code> object for those of the provided object.

Operators

OPERATOR	DESCRIPTION
operator=	Assigns the values to this <code>basic_istream</code> object from the object parameter.

Requirements

Header: `<sstream>`

Namespace: `std`

`basic_istream::allocator_type`

The type is a synonym for the template parameter `Alloc`.

```
typedef Alloc allocator_type;
```

`basic_istream::basic_istream`

Constructs an object of type `basic_istream`.

```
explicit basic_istream(
    ios_base::openmode _Mode = ios_base::in);

explicit basic_istream(
    const basic_string<Elem, Tr, Alloc>& str,
    ios_base::openmode _Mode = ios_base::in);

basic_istream(
    basic_istream&& right);
```

Parameters

_Mode

One of the enumerations in [ios_base::openmode](#).

str

An object of type `basic_string`.

right

An rvalue reference of a `basic_istream` object.

Remarks

The first constructor initializes the base class by calling [basic_istream](#)(`sb`), where `sb` is the stored object of class [basic_stringbuf](#)< `Elem`, `Tr`, `Alloc` >. It also initializes `sb` by calling [basic_stringbuf](#) < `Elem`, `Tr`, `Alloc` >(`_Mode` | `ios_base::in`).

The second constructor initializes the base class by calling [basic_istream](#)(`sb`). It also initializes `sb` by calling

```
basic_stringbuf < Elem, Tr, Alloc > ( str, _Mode | ios_base::in ).
```

The third constructor initializes the object with the contents of *right*, treated as an rvalue reference.

basic_istream::operator=

Assigns the values to this `basic_istream` object from the object parameter.

```
basic_istream& operator=(basic_istream&& right);
```

Parameters

right

An rvalue reference to a `basic_istream` object.

Remarks

The member operator replaces the contents of the object with the contents of *right*, treated as an rvalue reference move assignment.

basic_istream::rdbuf

Returns the address of the stored stream buffer of type `pointer` to `basic_stringbuf< Elem, Tr, Alloc >`.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer of type `pointer` to `basic_stringbuf< Elem, Tr, Alloc >`.

Example

See [basic_filebuf::close](#) for an example that uses `rdbuf`.

basic_istream::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str() const;

void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class `basic_string< Elem, Tr, Alloc >`, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns `rdbuf -> str`. The second member function calls `rdbuf -> str(_Newstr)`.

Example

See [basic_stringbuf::str](#) for an example that uses `str`.

basic_istream::swap

Exchanges the values of two `basic_istream` objects.

```
void swap(basic_istream& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	An <code>lvalue</code> reference to a <code>basic_istream</code> object.

Remarks

The member function exchanges the values of this object and the values of *right*.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_ostringstream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class [basic_stringbuf](#)< **Elem**, **Tr**, **Alloc** >.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_ostringstream : public basic_ostream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The class describes an object that controls insertion of elements and encoded objects into a stream buffer, with elements of type **Elem**, whose character traits are determined by the class **Tr**, and whose elements are allocated by an allocator of class **Alloc**. The object stores an object of class [basic_stringbuf](#)< **Elem**, **Tr**, **Alloc** >.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_ostringstream	Constructs an object of type basic_ostringstream .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	The type is a synonym for the template parameter <i>Alloc</i> .

Member functions

MEMBER FUNCTION	DESCRIPTION
rdbuf	Returns the address of the stored stream buffer of type pointer to basic_stringbuf < Elem , Tr , Alloc >.
str	Sets or gets the text in a string buffer without changing the write position.

Requirements

Header: <sstream>

Namespace: std

basic_ostringstream::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
typedef Alloc allocator_type;
```

basic_ostringstream::basic_ostringstream

Constructs an object of type basic_ostringstream.

```
explicit basic_ostringstream(ios_base::openmode _Mode = ios_base::out);

explicit basic_ostringstream(const basic_string<Elem, Tr, Alloc>& str, ios_base::openmode _Mode =
ios_base::out);
```

Parameters

_Mode

One of the enumerations in [ios_base::openmode](#).

str

An object of type `basic_string`.

Remarks

The first constructor initializes the base class by calling [basic_ostream](#)(**sb**), where `sb` is the stored object of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >. It also initializes **sb** by calling [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` > (`_Mode` | `ios_base::out`).

The second constructor initializes the base class by calling [basic_ostream](#)(**sb**). It also initializes `sb` by calling [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` > (`_Str`, `_Mode` | `ios_base::out`).

basic_ostringstream::rdbuf

Returns the address of the stored stream buffer of type `pointer` to [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer, of type `pointer` to [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Example

See [basic_filebuf::close](#) for an example that uses `rdbuf`.

basic_ostringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
basic_string<Elem, Tr, Alloc> str() const;

void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class `basic_string< Elem, Tr, Alloc >`, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns `rdbuf -> str`. The second member function calls `rdbuf -> str(_Newstr)`.

Example

See `basic_stringbuf::str` for an example that uses `str`.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

basic_stringstream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>>
class basic_stringstream : public basic_istream<Elem, Tr>
```

Parameters

Alloc

The allocator class.

Elem

The type of the basic element of the string.

Tr

The character traits specialized on the basic element of the string.

Remarks

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`. The object stores an object of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_stringstream	Constructs an object of type <code>basic_stringstream</code> .

Typedefs

TYPE NAME	DESCRIPTION
allocator_type	The type is a synonym for the template parameter <code>Alloc</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
rdbuf	Returns the address of the stored stream buffer of type <code>pointer</code> to basic_stringbuf < <code>Elem</code> , <code>Tr</code> , <code>Alloc</code> >.
str	Sets or gets the text in a string buffer without changing the write position.

Requirements

Header: <sstream>

Namespace: std

basic_stringstream::allocator_type

The type is a synonym for the template parameter `Alloc`.

```
typedef Alloc allocator_type;
```

basic_stringstream::basic_stringstream

Constructs an object of type `basic_stringstream`.

```
explicit basic_stringstream(ios_base::openmode _Mode = ios_base::in | ios_base::out);

explicit basic_stringstream(const basic_string<Elem, Tr, Alloc>& str, ios_base::openmode _Mode = ios_base::in
| ios_base::out);
```

Parameters

_Mode

One of the enumerations in [ios_base::openmode](#).

str

An object of type `basic_string`.

Remarks

The first constructor initializes the base class by calling [basic_istream](#)(**sb**), where `sb` is the stored object of class [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >. It also initializes `sb` by calling [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >(_Mode).

The second constructor initializes the base class by calling [basic_istream](#)(**sb**). It also initializes `sb` by calling [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >(_Str, _Mode).

basic_stringstream::rdbuf

Returns the address of the stored stream buffer of type **pointer** to [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer of type `pointer` to [basic_stringbuf](#)< **Elem**, **Tr**, `Alloc` >.

Example

See [basic_filebuf::close](#) for an example that uses `rdbuf`.

basic_stringstream::str

Sets or gets the text in a string buffer without changing the write position.


```
basic_string<Elem, Tr, Alloc> str() const;

void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr

The new string.

Return Value

Returns an object of class `basic_string< Elem, Tr, Alloc >`, whose controlled sequence is a copy of the sequence controlled by ***this**.

Remarks

The first member function returns `rdbuf -> str`. The second member function calls `rdbuf -> str(_Newstr)`.

Example

See `basic_stringbuf::str` for an example that uses `str`.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<stack>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines the template class `stack` and two supporting templates.

Syntax

```
#include <stack>
```

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the stack object on the left side of the operator is not equal to the stack object on the right side.
<code>operator<</code>	Tests if the stack object on the left side of the operator is less than the stack object on the right side.
<code>operator<=</code>	Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.
<code>operator==</code>	Tests if the stack object on the left side of the operator is equal to the stack object on the right side.
<code>operator></code>	Tests if the stack object on the left side of the operator is greater than the stack object on the right side.
<code>operator>=</code>	Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.

Classes

CLASS	DESCRIPTION
<code>stack Class</code>	A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<stack> operators

10/31/2018 • 7 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Tests if the stack object on the left side of the operator is not equal to stack object on the right side.

```
bool operator!=(const stack <Type, Container>& left, const stack <Type, Container>& right,);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stacks or stacks are not equal; **false** if stacks or stacks are equal.

Remarks

The comparison between stacks objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```

// stack_op_me.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with vector base containers
    stack <int, vector<int> > s1, s2, s3;

    // The following would have cause an error because stacks with
    // different base containers are not equality comparable
    // stack <int, list<int> > s3;

    s1.push( 1 );
    s2.push( 2 );
    s3.push( 1 );

    if ( s1 != s2 )
        cout << "The stacks s1 and s2 are not equal." << endl;
    else
        cout << "The stacks s1 and s2 are equal." << endl;

    if ( s1 != s3 )
        cout << "The stacks s1 and s3 are not equal." << endl;
    else
        cout << "The stacks s1 and s3 are equal." << endl;
}

```

```

The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.

```

operator<

Tests if the stack object on the left side of the operator is less than the stack object on the right side.

```
bool operator<(const stack <Type, Container>& left, const stack <Type, Container>& right);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stack on the left side of the operator is less than and not equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```

// stack_op_lt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with list base container
    stack<int, list<int> > s1, s2, s3;

    s1.push( 2 );
    s1.push( 4 );
    s1.push( 6 );
    s1.push( 8 );
    s2.push( 5 );
    s2.push( 10 );
    s3.push( 2 );
    s3.push( 4 );
    s3.push( 6 );
    s3.push( 8 );

    if ( s1 >= s2 )
        cout << "The stack s1 is greater than or equal to "
              << "the stack s2." << endl;
    else
        cout << "The stack s1 is less than "
              << "the stack s2." << endl;

    if ( s1 >= s3 )
        cout << "The stack s1 is greater than or equal to "
              << "the stack s3." << endl;
    else
        cout << "The stack s1 is less than "
              << "the stack s3." << endl;

    // to print out the stack s1 ( by unstacking the elements):
    stack<int>::size_type i_size_s1 = s1.size( );
    cout << "The stack s1 from the top down is: ( ";
    unsigned int i;
    for ( i = 1 ; i <= i_size_s1 ; i++ )
    {
        cout << s1.top( ) << " ";
        s1.pop( );
    }
    cout << ")." << endl;
}

```

The stack s1 is less than the stack s2.
 The stack s1 is greater than or equal to the stack s3.
 The stack s1 from the top down is: (8 6 4 2).

operator<=

Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.

```
bool operator<=(const stack<Type, Container>& left, const stack<Type, Container>& right);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stack on the left side of the operator is less than or equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_le.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with default deque base container
    stack <int> s1, s2, s3;

    s1.push( 5 );
    s1.push( 10 );
    s2.push( 1 );
    s2.push( 2 );
    s3.push( 5 );
    s3.push( 10 );

    if ( s1 <= s2 )
        cout << "The stack s1 is less than or equal to "
              << "the stack s2." << endl;
    else
        cout << "The stack s1 is greater than "
              << "the stack s2." << endl;

    if ( s1 <= s3 )
        cout << "The stack s1 is less than or equal to "
              << "the stack s3." << endl;
    else
        cout << "The stack s1 is greater than "
              << "the stack s3." << endl;
}
```

The stack s1 is greater than the stack s2.
The stack s1 is less than or equal to the stack s3.

operator==

Tests if the stack object on the left side of the operator is equal to stack object on the right side.

```
bool operator==(const stack <Type, Container>& left, const stack <Type, Container>& right);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stacks or stacks are equal; **false** if stacks or stacks are not equal.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// stack_op_eq.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with vector base containers
    stack <int, vector<int> > s1, s2, s3;

    // The following would have cause an error because stacks with
    // different base containers are not equality comparable
    // stack <int, list<int> > s3;

    s1.push( 1 );
    s2.push( 2 );
    s3.push( 1 );

    if ( s1 == s2 )
        cout << "The stacks s1 and s2 are equal." << endl;
    else
        cout << "The stacks s1 and s2 are not equal." << endl;

    if ( s1 == s3 )
        cout << "The stacks s1 and s3 are equal." << endl;
    else
        cout << "The stacks s1 and s3 are not equal." << endl;
}
```

The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.

operator>

Tests if the stack object on the left side of the operator is greater than the stack object on the right side.

```
bool operator>(const stack <Type, Container>& left, const stack <Type, Container>& right);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stack on the left side of the operator is greater than and not equal to the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_gt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with list base container
    stack <int, list<int> > s1, s2, s3;

    s1.push( 1 );
    s1.push( 2 );
    s1.push( 3 );
    s2.push( 5 );
    s2.push( 10 );
    s3.push( 1 );
    s3.push( 2 );

    if ( s1 > s2 )
        cout << "The stack s1 is greater than "
              << "the stack s2." << endl;
    else
        cout << "The stack s1 is not greater than "
              << "the stack s2." << endl;

    if ( s1 > s3 )
        cout << "The stack s1 is greater than "
              << "the stack s3." << endl;
    else
        cout << "The stack s1 is not greater than "
              << "the stack s3." << endl;
}
```

```
The stack s1 is not greater than the stack s2.
The stack s1 is greater than the stack s3.
```

operator>=

Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.


```
bool operator>=(const stack <Type, Container>& left, const stack <Type, Container>& right);
```

Parameters

left

An object of type `stack`.

right

An object of type `stack`.

Return Value

true if the stack on the left side of the operator is strictly less than the stack on the right side of the operator; otherwise **false**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

```
// stack_op_ge.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stacks with list base container
    stack <int, list<int> > s1, s2, s3;

    s1.push( 1 );
    s1.push( 2 );
    s2.push( 5 );
    s2.push( 10 );
    s3.push( 1 );
    s3.push( 2 );

    if ( s1 >= s2 )
        cout << "The stack s1 is greater than or equal to "
              << "the stack s2." << endl;
    else
        cout << "The stack s1 is less than "
              << "the stack s2." << endl;

    if ( s1 >= s3 )
        cout << "The stack s1 is greater than or equal to "
              << "the stack s3." << endl;
    else
        cout << "The stack s1 is less than "
              << "the stack s3." << endl;
}
```

The stack s1 is less than the stack s2.
The stack s1 is greater than or equal to the stack s3.

See also

<stack>

stack Class

10/31/2018 • 8 minutes to read • [Edit Online](#)

A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type. The stack class is used when it is important to be clear that only stack operations are being performed on the container.

Syntax

```
template <class Type, class Container= deque <Type>>
class stack
```

Parameters

Type

The element data type to be stored in the stack.

Container

The type of the underlying container used to implement the stack. The default value is the class `deque <Type>`.

Remarks

The elements of class `Type` stipulated in the first template parameter of a stack object are synonymous with [value_type](#) and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for stack include [deque](#), [list class](#), and [vector class](#), or any other sequence container that supports the operations of `back`, `push_back`, and `pop_back`. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The stack objects are equality comparable if and only if the elements of class `Type` are equality comparable and are less-than comparable if and only if the elements of class `Type` are less-than comparable.

- The stack class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
- The [queue class](#) supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the queue class.
- The [priority_queue class](#) orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Constructors

CONSTRUCTOR	DESCRIPTION
stack	Constructs a <code>stack</code> that is empty or that is a copy of a base container object.

Typedefs

TYPE NAME	DESCRIPTION
container_type	A type that provides the base container to be adapted by a <code>stack</code> .
size_type	An unsigned integer type that can represent the number of elements in a <code>stack</code> .
value_type	A type that represents the type of object stored as an element in a <code>stack</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
empty	Tests if the <code>stack</code> is empty.
pop	Removes the element from the top of the <code>stack</code> .
push	Adds an element to the top of the <code>stack</code> .
size	Returns the number of elements in the <code>stack</code> .
top	Returns a reference to an element at the top of the <code>stack</code> .

Requirements

Header: `<stack>`

Namespace: `std`

`stack::container_type`

A type that provides the base container to be adapted.

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. All three C++ Standard Library sequence container classes — the vector class, list class, and the default class deque — meet the requirements to be used as the base container for a stack object. User-defined types satisfying these requirements may also be used.

For more information on `Container`, see the Remarks section of the [stack Class](#) topic.

Example

See the example for [stack::stack](#) for an example of how to declare and use `container_type`.

stack::empty

Tests if a stack is empty.

```
bool empty() const;
```

Return Value

true if the stack is empty; **false** if the stack is nonempty.

Example

```
// stack_empty.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    // Declares stacks with default deque base container
    stack <int> s1, s2;

    s1.push( 1 );

    if ( s1.empty( ) )
        cout << "The stack s1 is empty." << endl;
    else
        cout << "The stack s1 is not empty." << endl;

    if ( s2.empty( ) )
        cout << "The stack s2 is empty." << endl;
    else
        cout << "The stack s2 is not empty." << endl;
}
```

```
The stack s1 is not empty.
The stack s2 is empty.
```

stack::pop

Removes the element from the top of the stack.

```
void pop();
```

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```

// stack_pop.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    stack <int> s1, s2;

    s1.push( 10 );
    s1.push( 20 );
    s1.push( 30 );

    stack <int>::size_type i;
    i = s1.size( );
    cout << "The stack length is " << i << "." << endl;

    i = s1.top( );
    cout << "The element at the top of the stack is "
        << i << "." << endl;

    s1.pop( );

    i = s1.size( );
    cout << "After a pop, the stack length is "
        << i << "." << endl;

    i = s1.top( );
    cout << "After a pop, the element at the top of the stack is "
        << i << "." << endl;
}

```

```

The stack length is 3.
The element at the top of the stack is 30.
After a pop, the stack length is 2.
After a pop, the element at the top of the stack is 20.

```

stack::push

Adds an element to the top of the stack.

```
void push(const Type& val);
```

Parameters

val

The element added to the top of the stack.

Remarks

The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

```
// stack_push.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    stack <int> s1;

    s1.push( 10 );
    s1.push( 20 );
    s1.push( 30 );

    stack <int>::size_type i;
    i = s1.size( );
    cout << "The stack length is " << i << "." << endl;

    i = s1.top( );
    cout << "The element at the top of the stack is "
        << i << "." << endl;
}
```

The stack length is 3.
The element at the top of the stack is 30.

stack::size

Returns the number of elements in the stack.

```
size_type size() const;
```

Return Value

The current length of the stack.

Example

```
// stack_size.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    stack <int> s1, s2;
    stack <int>::size_type i;

    s1.push( 1 );
    i = s1.size( );
    cout << "The stack length is " << i << "." << endl;

    s1.push( 2 );
    i = s1.size( );
    cout << "The stack length is now " << i << "." << endl;
}
```

```
The stack length is 1.  
The stack length is now 2.
```

stack::size_type

An unsigned integer type that can represent the number of elements in a stack.

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for `size_type` of the base container adapted by the stack.

Example

See the example for [size](#) for an example of how to declare and use `size_type`.

stack::stack

Constructs a stack that is empty or that is a copy of a base container class.

```
stack();  
  
explicit stack(const container_type& right);
```

Parameters

right

The container of which the constructed stack is to be a copy.

Example


```

// stack_stack.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <list>
#include <iostream>

int main( )
{
    using namespace std;

    // Declares stack with default deque base container
    stack<char> dsc1;

    //Explicitly declares a stack with deque base container
    stack<char, deque<char> > dsc2;

    // Declares a stack with vector base containers
    stack<int, vector<int> > vsi1;

    // Declares a stack with list base container
    stack<int, list<int> > lsi;

    // The second member function copies elements from a container
    vector<int> v1;
    v1.push_back( 1 );
    stack<int, vector<int> > vsi2( v1 );
    cout << "The element at the top of stack vsi2 is "
         << vsi2.top( ) << "." << endl;
}

```

The element at the top of stack vsi2 is 1.

stack::top

Returns a reference to an element at the top of the stack.

```

reference top();

const_reference top() const;

```

Return Value

A reference to the last element in the container at the top of the stack.

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

If the return value of `top` is assigned to a `const_reference`, the stack object cannot be modified. If the return value of `top` is assigned to a `reference`, the stack object can be modified.

Example

```
// stack_top.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    stack<int> s1;

    s1.push( 1 );
    s1.push( 2 );

    int& i = s1.top( );
    const int& ii = s1.top( );

    cout << "The top integer of the stack s1 is "
         << i << "." << endl;
    i--;
    cout << "The next integer down is "<< ii << "." << endl;
}
```

The top integer of the stack s1 is 2.
The next integer down is 1.

stack::value_type

A type that represents the type of object stored as an element in a stack.

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for `value_type` of the base container adapted by the stack.

Example

```
// stack_value_type.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
    using namespace std;
    // Declares stacks with default deque base container
    stack<int>::value_type AnInt;

    AnInt = 69;
    cout << "The value_type is AnInt = " << AnInt << endl;

    stack<int> s1;
    s1.push( AnInt );
    cout << "The element at the top of the stack is "
         << s1.top( ) << "." << endl;
}
```

The value_type is AnInt = 69
The element at the top of the stack is 69.

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<stdexcept>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines several standard classes used for reporting exceptions. The classes form a derivation hierarchy all derived from class [exception](#) and include two general types of exceptions: logical errors and run-time errors. The logical errors are caused programmer mistakes. They derive from the base class `logic_error` and include:

- `domain_error`
- `invalid_argument`
- `length_error`
- `out_of_range`

The run-time errors occur because of mistakes in either the library functions or in the run-time system. They derive from the base class `runtime_error` and include:

- `overflow_error`
- `range_error`
- `underflow_error`

Classes

CLASS	DESCRIPTION
domain_error Class	The class serves as the base class for all exceptions thrown to report a domain error.
invalid_argument Class	The class serves as the base class for all exceptions thrown to report an invalid argument.
length_error Class	The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.
logic_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.
out_of_range Class	The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.
overflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic overflow.
range_error Class	The class serves as the base class for all exceptions thrown to report a range error.
runtime_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.

CLASS	DESCRIPTION
underflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

domain_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report a domain error.

Syntax

```
class domain_error : public logic_error {
public:
    explicit domain_error(const string& message);

    explicit domain_error(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message** `.data`.

Example

```
// domain_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
    try
    {
        throw domain_error( "Your domain is in error!" );
    }
    catch (exception &e)
    {
        cerr << "Caught: " << e.what( ) << endl;
        cerr << "Type: " << typeid(e).name( ) << endl;
    }
};

/* Output:
Caught: Your domain is in error!
Type: class std::domain_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[logic_error Class](#)

[Thread Safety in the C++ Standard Library](#)

invalid_argument Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report an invalid argument.

Syntax

```
class invalid_argument : public logic_error {
public:
    explicit invalid_argument(const string& message);

    explicit invalid_argument(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message** [data](#).

Example

```
// invalid_arg.cpp
// compile with: /EHsc /GR
#include <bitset>
#include <iostream>

using namespace std;

int main( )
{
    try
    {
        bitset< 32 > bitset( string( "11001010101100001b100101010110000" ) );
    }
    catch ( exception &e )
    {
        cerr << "Caught " << e.what( ) << endl;
        cerr << "Type " << typeid( e ).name( ) << endl;
    }
};

/* Output:
Caught invalid bitset<N> char
Type class std::invalid_argument
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[logic_error Class](#)

length_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.

Syntax

```
class length_error : public logic_error {  
public:  
    explicit length_error(const string& message);  
  
    explicit length_error(const char *message);  
  
};
```

Remarks

The value returned by [what](#) is a copy of **message**.[data](#).

Example

```

// length_error.cpp
// compile with: /EHsc /GR /MDd
#include <vector>
#include <iostream>

using namespace std;

template<class T>
class stingyallocator : public allocator< T>
{
public:
    template <class U>
        struct rebind { typedef stingyallocator<U> other; };
    _SIZT max_size( ) const
    {
        return 10;
    };
};

int main( )
{
    try
    {
        vector<int, stingyallocator< int > > myv;
        for ( int i = 0; i < 11; i++ ) myv.push_back( i );
    }
    catch ( exception &e )
    {
        cerr << "Caught " << e.what( ) << endl;
        cerr << "Type " << typeid( e ).name( ) << endl;
    };
}
/* Output:
Caught vector<T> too long
Type class std::length_error
*/

```

Requirements

Header: <stdexcept>

Namespace: std

See also

[logic_error Class](#)

[Thread Safety in the C++ Standard Library](#)

logic_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.

Syntax

```
class logic_error : public exception {
public:
    explicit logic_error(const string& message);

    explicit logic_error(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message**.[data](#).

Example

```
// logic_error.cpp
// compile with: /EHsc /GR
#include <iostream>
using namespace std;

int main( )
{
    try
    {
        throw logic_error( "logic error" );
    }
    catch ( exception &e )
    {
        cerr << "Caught: " << e.what( ) << endl;
        cerr << "Type: " << typeid( e ).name( ) << endl;
    };
}
```

```
Caught: logic error
Type: class std::logic_error
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[exception Class](#)

out_of_range Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.

Syntax

```
class out_of_range : public logic_error {
public:
    explicit out_of_range(const string& message);

    explicit out_of_range(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message**. [data](#).

Example

```
// out_of_range.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

using namespace std;

int main() {
    // out_of_range
    try {
        string str( "Micro" );
        string rstr( "soft" );
        str.append( rstr, 5, 3 );
        cout << str << endl;
    }
    catch ( exception &e ) {
        cerr << "Caught: " << e.what( ) << endl;
    };
}
```

Output

```
Caught: invalid string position
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[logic_error Class](#)

[Thread Safety in the C++ Standard Library](#)

overflow_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report an arithmetic overflow.

Syntax

```
class overflow_error : public runtime_error {
public:
    explicit overflow_error(const string& message);

    explicit overflow_error(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message**. [data](#).

Example

```
// overflow_error.cpp
// compile with: /EHsc /GR
#include <bitset>
#include <iostream>

using namespace std;

int main( )
{
    try
    {
        bitset< 33 > bitset;
        bitset[32] = 1;
        bitset[0] = 1;
        unsigned long x = bitset.to_ulong( );
    }
    catch ( exception &e )
    {
        cerr << "Caught " << e.what( ) << endl;
        cerr << "Type " << typeid( e ).name( ) << endl;
    }
};

/* Output:
Caught bitset<N> overflow
Type class std::overflow_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[runtime_error](#) Class

[Thread Safety in the C++ Standard Library](#)

range_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report a range error.

Syntax

```
class range_error : public runtime_error {
public:
    explicit range_error(const string& message);
    explicit range_error(const char *message);
};
```

Remarks

The value returned by [what](#) is a copy of `message.data`. For more information, see [basic_string::data](#).

Example

```
// range_error.cpp
// compile with: /EHsc /GR
#include <iostream>
using namespace std;
int main()
{
    try
    {
        throw range_error( "The range is in error!" );
    }
    catch (range_error &e)
    {
        cerr << "Caught: " << e.what( ) << endl;
        cerr << "Type: " << typeid( e ).name( ) << endl;
    };
}
/* Output:
Caught: The range is in error!
Type: class std::range_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[runtime_error Class](#)

[Thread Safety in the C++ Standard Library](#)

runtime_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.

Syntax

```
class runtime_error : public exception {
public:
    explicit runtime_error(const string& message);

    explicit runtime_error(const char *message);

};
```

Remarks

The value returned by [exception Class](#) is a copy of **message** `.data`.

Example

```
// runtime_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
    // runtime_error
    try
    {
        locale loc( "test" );
    }
    catch ( exception &e )
    {
        cerr << "Caught " << e.what( ) << endl;
        cerr << "Type " << typeid( e ).name( ) << endl;
    };
}

/* Output:
Caught bad locale name
Type class std::runtime_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

exception Class

Thread Safety in the C++ Standard Library

underflow_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

Syntax

```
class underflow_error : public runtime_error {
public:
    explicit underflow_error(const string& message);

    explicit underflow_error(const char *message);

};
```

Remarks

The value returned by [what](#) is a copy of **message** [.data](#).

Example

```
// underflow_error.cpp
// compile with: /EHsc /GR
#include <iostream>

using namespace std;

int main( )
{
    try
    {
        throw underflow_error( "The number's a bit small, captain!" );
    }
    catch ( exception &e ) {
        cerr << "Caught: " << e.what( ) << endl;
        cerr << "Type: " << typeid( e ).name( ) << endl;
    }
};

/* Output:
Caught: The number's a bit small, captain!
Type: class std::underflow_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

[runtime_error Class](#)

[Thread Safety in the C++ Standard Library](#)

<streambuf>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Include the iostreams standard header `<streambuf>` to define the template class `basic_streambuf`, which is basic to the operation of the iostreams classes. This header is typically included for you by another of the iostreams headers; you rarely need to include it directly.

Syntax

```
#include <streambuf>
```

Typedefs

TYPE NAME	DESCRIPTION
<code>streambuf</code>	A specialization of <code>basic_streambuf</code> that uses char as the template parameters.
<code>wstreambuf</code>	A specialization of <code>basic_streambuf</code> that uses wchar_t as the template parameters.

Classes

CLASS	DESCRIPTION
<code>basic_streambuf</code> Class	The template class describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<streambuf> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

streambuf	wstreambuf

streambuf

A specialization of `basic_streambuf` that uses **char** as the template parameters.

```
typedef basic_streambuf<char, char_traits<char>> streambuf;
```

Remarks

The type is a synonym for the template class [basic_streambuf](#), specialized for elements of type **char** with default character traits.

wstreambuf

A specialization of `basic_streambuf` that uses **wchar_t** as the template parameters.

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t>> wstreambuf;
```

Remarks

The type is a synonym for the template class [basic_streambuf](#), specialized for elements of type **wchar_t** with default character traits.

See also

[<streambuf>](#)

basic_streambuf Class

11/9/2018 • 25 minutes to read • [Edit Online](#)

Describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

Syntax

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_streambuf;
```

Parameters

Elem

A [char_type](#).

Tr

The character [traits_type](#).

Remarks

The template class describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream. An object of class `basic_streambuf` helps control a stream with elements of type *Tr*, also known as [char_type](#), whose character traits are determined by the class [char_traits](#), also known as [traits_type](#).

Every stream buffer conceptually controls two independent streams: one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a `basic_stringbuf< Elem, Tr >` object, for example, is what you later extract from its input stream. When you position one stream of a `basic_filebuf< Elem, Tr >` object, you position the other stream in tandem.

The public interface to template class `basic_streambuf` supplies the operations that are common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each derived stream buffer in this library describes how it specializes the behavior of its protected virtual member functions. The default behavior for the base class, which is often to do nothing, is described in this topic.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An input buffer, for example, is characterized by:

- [eback](#), a pointer to the beginning of the buffer.
- [gptr](#), a pointer to the next element to read.
- [egptr](#), a pointer just past the end of the buffer.

Similarly, an output buffer is characterized by:

- [pbase](#), a pointer to the beginning of the buffer.
- [pptr](#), a pointer to the next element to write.

- [epptr](#), a pointer just past the end of the buffer.

For any buffer, the following protocol is used:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. They can be safely compared for order.
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the write position designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the read position designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the putback position designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from `basic_streambuf < Elem, Tr >` must cooperate in maintaining this protocol.

An object of class `basic_streambuf < Elem, Tr >` stores the six pointers previously described. It also stores a locale object in an object of type [locale](#) for potential use by a derived stream buffer.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_streambuf	Constructs an object of type <code>basic_streambuf</code> .

Typedefs

TYPE NAME	DESCRIPTION
char_type	Associates a type name with the <code>Elem</code> template parameter.
int_type	Associates a type name within <code>basic_streambuf</code> scope with the <code>Elem</code> template parameter.
off_type	Associates a type name within <code>basic_streambuf</code> scope with the <code>Elem</code> template parameter.
pos_type	Associates a type name within <code>basic_streambuf</code> scope with the <code>Elem</code> template parameter.
traits_type	Associates a type name with the <code>Tr</code> template parameter.

Member functions

MEMBER FUNCTION	DESCRIPTION
eback	A protected function that returns a pointer to the beginning of the input buffer.
egptr	A protected function that returns a pointer just past the end of the input buffer.

MEMBER FUNCTION	DESCRIPTION
epptr	A protected function that returns a pointer just past the end of the output buffer.
gbump	A protected function that adds <code>count</code> to the next pointer for the input buffer.
getloc	Gets the <code>basic_streambuf</code> object's locale.
gptr	A protected function that returns a pointer to the next element of the input buffer.
imbue	A protected, virtual function called by pubimbue .
in_avail	Returns the number of elements that are ready to be read from the buffer.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	A protected virtual member function that tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).
pbase	A protected function that returns a pointer to the beginning of the output buffer.
pbump	A protected function that adds <code>count</code> to the next pointer for the output buffer.
pptr	A protected function that returns a pointer to the next element of the output buffer.
pubimbue	Sets the <code>basic_streambuf</code> object's locale.
pubseekoff	Calls seekoff , a protected virtual function that is overridden in a derived class.
pubseekpos	Calls seekpos , a protected virtual function that is overridden in a derived class and resets the current pointer position.
pubsetbuf	Calls setbuf , a protected virtual function that is overridden in a derived class.
pubsync	Calls sync , a protected virtual function that is overridden in a derived class and updates the external stream associated with this buffer.
sbumpc	Reads and returns the current element, moving the stream pointer.
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.

MEMBER FUNCTION	DESCRIPTION
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
setbuf	The protected virtual member function performs an operation particular to each derived stream buffer.
setg	A protected function that stores <code>_Gbeg</code> in the beginning pointer, <code>_Gnext</code> in the next pointer, and <code>_Gend</code> in the end pointer for the input buffer.
setp	A protected function that stores <code>_Pbeg</code> in the beginning pointer and <code>_Pend</code> in the end pointer for the output buffer.
sgetc	Returns current element without changing position in the stream.
sgetn	Returns the number of elements read.
showmanyc	Protected virtual member function that returns a count of the number of characters that can be extracted from the input stream and ensure that the program will not be subject to an indefinite wait.
snextc	Reads the current element and returns the following element.
sputbackc	Puts a <code>char_type</code> in the stream.
sputc	Puts a character into the stream.
sputn	Puts a character string into the stream.
stossc	Move past the current element in the stream.
sungetc	Gets a character from the stream.
swap	Exchanges the values in this object for the values in the provided <code>basic_streambuf</code> object parameter.
sync	A protected virtual function that tries to synchronize the controlled streams with any associated external streams.
uflow	A protected virtual function that extracts the current element from the input stream.
underflow	A protected virtual function that extracts the current element from the input stream.
xsgetn	A protected virtual function that extracts elements from the input stream.
xsputn	A protected virtual function that inserts elements into the output stream.

Operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns the values of this object from another <code>basic_streambuf</code> object.

Requirements

Header: <streambuf>

Namespace: std

basic_streambuf::basic_streambuf

Constructs an object of type `basic_streambuf`.

```
basic_streambuf();

basic_streambuf(const basic_streambuf& right);
```

Parameters

right

An lvalue reference to the `basic_streambuf` object that is used to set the values for this `basic_streambuf` object.

Remarks

The first protected constructor stores a null pointer in all pointers controlling the input buffer and the output buffer. It also stores `locale::classic` in the locale object. For more information, see [locale::classic](#).

The second protected constructor copies the pointers and locale from *right*.

basic_streambuf::char_type

Associates a type name with the **Elem** template parameter.

```
typedef Elem char_type;
```

basic_streambuf::eback

A protected function that returns a pointer to the beginning of the input buffer.

```
char_type *eback() const;
```

Return Value

A pointer to the beginning of the input buffer.

basic_streambuf::egptr

A protected function that returns a pointer just past the end of the input buffer.

```
char_type *egptr() const;
```

Return Value

A pointer just past the end of the input buffer.

basic_streambuf::epptr

A protected function that returns a pointer just past the end of the output buffer.

```
char_type *epptr() const;
```

Return Value

A pointer just past the end of the output buffer.

basic_streambuf::gbump

A protected function that adds *count* to the next pointer for the input buffer.

```
void gbump(int count);
```

Parameters

count

The amount by which to advance the pointer.

basic_streambuf::getloc

Gets the `basic_streambuf` object's locale.

```
locale getloc() const;
```

Return Value

The stored locale object.

Remarks

For related information, see [ios_base::getloc](#).

Example

```
// basic_streambuf_getloc.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << cout.rdbuf( )->getloc( ).name( ).c_str( ) << endl;
}
```

```
c
```

basic_streambuf::gpptr

A protected function that returns a pointer to the next element of the input buffer.

```
char_type *gptr() const;
```

Return Value

A pointer to the next element of the input buffer.

basic_streambuf::imbue

A protected virtual function called by [pubimbue](#).

```
virtual void imbue(const locale& _Loc);
```

Parameters

_Loc

A reference to a locale.

Remarks

The default behavior is to do nothing.

basic_streambuf::in_avail

Returns the number of elements that are ready to be read from the buffer.

```
streamsize in_avail();
```

Return Value

The number of elements that are ready to be read from the buffer.

Remarks

If a [read position](#) is available, the member function returns [egptr](#) - [gptr](#). Otherwise, it returns [showmanyc](#).

Example

```
// basic_streambuf_in_avail.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    char c;
    // cin's buffer is empty, in_avail will return 0
    cout << cin.rdbuf( )->in_avail( ) << endl;
    cin >> c;
    cout << cin.rdbuf( )->in_avail( ) << endl;
}
```

basic_streambuf::int_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::int_type int_type;
```

basic_streambuf::off_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::off_type off_type;
```

basic_streambuf::operator=

Assigns the values of this object from another `basic_streambuf` object.

```
basic_streambuf& operator=(const basic_streambuf& right);
```

Parameters

right

An lvalue reference to the `basic_streambuf` object that is used to assign values to this object.

Remarks

The protected member operator copies from *right* the pointers that control the input buffer and the output buffer. It also stores `right.getloc()` in the `locale` object. It returns `*this`.

basic_streambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int_type overflow(int_type _Meta = traits_type::eof());
```

Parameters

_Meta

The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof` or throws an exception. Otherwise, it returns `traits_type::not_eof(_Meta)`. The default behavior is to return `traits_type::eof`.

Remarks

If *_Meta* does not compare equal to `traits_type::eof`, the protected virtual member function endeavors to insert the element `traits_type::to_char_type(_Meta)` into the output stream. It can do so in various ways:

- If a `write position` is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

The virtual overflow function, together with the `sync` and `underflow` functions, defines the characteristics of the streambuf-derived class. Each derived class might implement overflow differently, but the interface with the calling stream class is the same.

The `overflow` function is most frequently called by public `streambuf` functions like `sputc` and `sputn` when the put area is full, but other classes, including the stream classes, can call `overflow` anytime.

The function consumes the characters in the put area between the `pbase` and `pptr` pointers and then reinitializes the put area. The `overflow` function must also consume `nCh` (if `nCh` is not `EOF`), or it might choose to put that character in the new put area so that it will be consumed on the next call.

The definition of consume varies among derived classes. For example, the `filebuf` class writes its characters to a file, while the `strstreambuf` class keeps them in its buffer and (if the buffer is designated as dynamic) expands the buffer in response to a call to `overflow`. This expansion is achieved by freeing the old buffer and replacing it with a new, larger one. The pointers are adjusted as necessary.

basic_streambuf::pbackfail

A protected virtual member function that tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
virtual int_type pbackfail(int_type _Meta = traits_type::eof());
```

Parameters

_Meta

The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function cannot succeed, it returns `traits_type::eof` or throws an exception. Otherwise, it returns some other value. The default behavior is to return `traits_type::eof`.

Remarks

If *_Meta* compares equal to `traits_type::eof`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `traits_type::to_char_type(_Meta)`. The function can put back an element in various ways:

- If a putback position is available, it can store the element into the putback position and decrement the next pointer for the input buffer.
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

basic_streambuf::pbase

A protected function that returns a pointer to the beginning of the output buffer.

```
char_type *pbase() const;
```

Return Value

A pointer to the beginning of the output buffer.

basic_streambuf::pbump

A protected function that adds *count* to the next pointer for the output buffer.

```
void pbump(int count);
```


Parameters

count

The number of characters by which to move the write position forward.

basic_streambuf::pos_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
typedef typename traits_type::pos_type pos_type;
```

basic_streambuf::pptr

A protected function that returns a pointer to the next element of the output buffer.

```
char_type *pptr() const;
```

Return Value

A pointer to the next element of the output buffer.

basic_streambuf::pubimbue

Sets the basic_streambuf object's locale.

```
locale pubimbue(const locale& _Loc);
```

Parameters

_Loc

A reference to a locale.

Return Value

The previous value stored in the locale object.

Remarks

The member function stores *_Loc* in the locale object and calls [imbue](#).

Example

See [basic_ios::imbue](#) for an example that uses `pubimbue`.

basic_streambuf::pubseekoff

Calls [seekoff](#), a protected virtual function that is overridden in a derived class.

```
pos_type pubseekoff(off_type _Off,  
    ios_base::seekdir _Way,  
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See [seekdir](#) for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position ([seekoff](#)(*_Off*, *_Way*, *_Which*)).

Remarks

Moves the pointer relative to *_Way*.

basic_streambuf::pubseekpos

Calls [seekpos](#), a protected virtual function that is overridden in a derived class, and resets the current pointer position.

```
pos_type pubseekpos(pos_type _Sp, ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The member function returns [seekpos](#)(*_Sp*, *_Which*).

basic_streambuf::pubsetbuf

Calls [setbuf](#), a protected virtual function that is overridden in a derived class.

```
basic_streambuf<Elem, Tr> *pubsetbuf(  
    char_type* _Buffer,  
    streamsize count);
```

Parameters

_Buffer

A pointer to `char_type` for this instantiation.

count

The size of the buffer.

Return Value

Returns [setbuf](#)(*_Buffer*, *count*).

basic_streambuf::pubsync

Calls [sync](#), a protected virtual function that is overridden in a derived class, and updates the external stream associated with this buffer.

```
int pubsync();
```

Return Value

Returns [sync](#) or -1 if failure.

basic_streambuf::sbumpc

Reads and returns the current element, moving the stream pointer.

```
int_type sbumpc();
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::to_int_type(*gptr)** and increments the next pointer for the input buffer. Otherwise, it returns [uflow](#).

Example

```
// basic_streambuf_sbumpc.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int i = 0;
    i = cin.rdbuf( )->sbumpc( );
    cout << i << endl;
}
```

```
3
```

```
33
51
```

basic_streambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual pos_type seekoff(
    off_type _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See [seekdir](#) for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position (`seekoff` (`_Off`, `_Way`, `_Which`)).

Remarks

The new position is determined as follows:

- If `_Way` == `ios_base::beg`, the new position is the beginning of the stream plus `_Off`.
- If `_Way` == `ios_base::cur`, the new position is the current stream position plus `_Off`.
- If `_Way` == `ios_base::end`, the new position is the end of the stream plus `_Off`.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual pos_type seekpos(pos_type _Sp, ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Sp

The position to seek for.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position, or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The new position is `_Sp`.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position (-1). The default behavior is to return an invalid stream position.

basic_streambuf::setbuf

A protected virtual member function that performs an operation particular to each derived stream buffer.

```
virtual basic_streambuf<Elem, Tr> *setbuf(
    char_type* _Buffer,
    streamsize count);
```

Parameters

_Buffer

Pointer to a buffer.

count

Size of the buffer.

Return Value

The default behavior is to return **this**.

Remarks

See [basic_filebuf](#). `setbuf` provides an area of memory for the `streambuf` object to use. How the buffer is used in defined in the derived classes.

basic_streambuf::setg

A protected function that stores *_Gbeg* in the beginning pointer, `_Gnext` in the next pointer, and `_Gend` in the end pointer for the input buffer.

```
void setg(char_type* _Gbeg,
    char_type* _Gnext,
    char_type* _Gend);
```

Parameters

_Gbeg

A pointer to the beginning of the buffer.

_Gnext

A pointer to somewhere in the middle of the buffer.

_Gend

A pointer to the end of the buffer.

basic_streambuf::setp

A protected function that stores *_Pbeg* in the beginning pointer and *_Pend* in the end pointer for the output buffer.

```
void setp(char_type* _Pbeg, char_type* _Pend);
```

Parameters

_Pbeg

A pointer to the beginning of the buffer.

_Pend

A pointer to the end of the buffer.

basic_streambuf::sgetc

Returns current element without changing position in the stream.

```
int_type sgetc();
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::to_int_type**(`*gptr`). Otherwise, it returns [underflow](#).

Example

```
// basic_streambuf_sgetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    ifstream myfile( "basic_streambuf_sgetc.txt", ios::in );

    char i = myfile.rdbuf( )->sgetc( );
    cout << i << endl;
    i = myfile.rdbuf( )->sgetc( );
    cout << i << endl;
}
```

basic_streambuf::sgetn

Extracts up to *count* characters from the input buffer and stores them in the provided buffer *ptr*.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
streamsize sgetn(
    char_type* ptr,
    streamsize count);
```

Parameters

ptr

The buffer to contain the extracted characters.

count

The number of elements to read.

Return Value

The number of elements read. See [streamsize](#) for more information.

Remarks

The member function returns [xsgetn](#)(`ptr`, `count`).

Example

```

// basic_streambuf_sgetn.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    ifstream myfile("basic_streambuf_sgetn.txt", ios::in);
    char a[10];

    // Extract 3 characters from myfile and store them in a.
    streamsize i = myfile.rdbuf()->sgetn(&a[0], 3); // C4996
    a[i] = myfile.widen('\0');

    // Display the size and contents of the buffer passed to sgetn.
    cout << i << " " << a << endl;

    // Display the contents of the original input buffer.
    cout << myfile.rdbuf() << endl;
}

```

basic_streambuf::showmanyc

A protected virtual member function that returns a count of the number of characters that can be extracted from the input stream and ensure that the program will not be subject to an indefinite wait.

```
virtual streamsize showmanyc();
```

Return Value

The default behavior is to return zero.

basic_streambuf::snextc

Reads the current element and returns the following element.

```
int_type snextc();
```

Return Value

The next element in the stream.

Remarks

The member function calls [sbumpc](#) and, if that function returns **traits_type::eof**, returns **traits_type::eof**. Otherwise, it returns [sgetc](#).

Example

```
// basic_streambuf_snextc.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
    using namespace std;
    int i = 0;
    i = cin.rdbuf( )->snextc( );
    // cout << ( int )char_traits<char>::eof << endl;
    cout << i << endl;
}
```

aa

aa97

basic_streambuf::sputbackc

Puts a `char_type` in the stream.

```
int_type sputbackc(char_type _Ch);
```

Parameters

_Ch

The character.

Return Value

Returns the character or failure.

Remarks

If a putback position is available and *_Ch* compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns **traits_type::to_int_type**(`_Ch`). Otherwise, it returns **backfail**(`_Ch`).

Example


```
// basic_streambuf_sputbackc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;

    ifstream myfile("basic_streambuf_sputbackc.txt",
        ios::in);

    int i = myfile.rdbuf()->sgetc();
    cout << (char)i << endl;
    int j = myfile.rdbuf()->sputbackc('z');
    if (j == 'z')
    {
        cout << "it worked" << endl;
    }
    i = myfile.rdbuf()->sgetc();
    cout << (char)i << endl;
}
```

basic_streambuf::sputc

Puts a character into the stream.

```
int_type sputc(char_type _Ch);
```

Parameters

_Ch

The character.

Return Value

Returns the character, if successful.

Remarks

If a `write position` is available, the member function stores *_Ch* in the write position, increments the next pointer for the output buffer, and returns **traits_type::to_int_type**(`_ch`). Otherwise, it returns `overflow`(`_ch`).

Example

```
// basic_streambuf_sputc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;

    int i = cout.rdbuf( )->sputc( 'a' );
    cout << endl << ( char )i << endl;
}
```

```
a
a
```

basic_streambuf::sputn

Puts a character string into the stream.

```
streamsize sputn(const char_type* ptr, streamsize count);
```

Parameters

ptr

The character string.

count

The count of characters.

Return Value

The number of characters actually inserted into the stream.

Remarks

The member function returns [xsputn](#)(`ptr`, `count`). See the Remarks section of this member for more information.

Example

```
// basic_streambuf_sputn.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    streamsize i = cout.rdbuf()->sputn("test", 4);
    cout << endl << i << endl;
}
```

```
test
4
```

basic_streambuf::stossc

Move past the current element in the stream.

```
void stossc();
```

Remarks

The member function calls [sbumpc](#). Note that an implementation is not required to supply this member function.

Example

```
// basic_streambuf_stossc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    ifstream myfile( "basic_streambuf_stossc.txt", ios::in );

    myfile.rdbuf( )->stossc( );
    char i = myfile.rdbuf( )->sgetc( );
    cout << i << endl;
}
```

basic_streambuf::sungetc

Gets a character from the stream.

```
int_type sungetc();
```

Return Value

Returns either the character or failure.

Remarks

If a putback position is available, the member function decrements the next pointer for the input buffer and returns `traits_type::to_int_type(*gptr)`. However, it is not always possible to determine the last character read so that it can be captured in the state of the current buffer. If this is true, then the function returns `ebackfail`. To avoid this situation, keep track of the character to put back and call `sputbackc(ch)`, which will not fail provided you don't call it at the beginning of the stream and you don't try to put back more than one character.

Example

```

// basic_streambuf_sungetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;

    ifstream myfile( "basic_streambuf_sungetc.txt", ios::in );

    // Read and increment
    int i = myfile.rdbuf( )->sgetc( );
    cout << ( char )i << endl;

    // Read and increment
    i = myfile.rdbuf( )->sgetc( );
    cout << ( char )i << endl;

    // Decrement, read, and do not increment
    i = myfile.rdbuf( )->sgetc( );
    cout << ( char )i << endl;

    i = myfile.rdbuf( )->sgetc( );
    cout << ( char )i << endl;

    i = myfile.rdbuf( )->sgetc( );
    cout << ( char )i << endl;
}

```

basic_streambuf::swap

Exchanges the values in this object for the values in the provided `basic_streambuf` object.

```
void swap(basic_streambuf& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	An lvalue reference to the <code>basic_streambuf</code> object that is used to exchange values.

Remarks

The protected member function exchanges with *right* all the pointers controlling the `input buffer` and the `output buffer`. It also exchanges `right.getloc()` with the `locale` object.

basic_streambuf::sync

A protected virtual function that tries to synchronize the controlled streams with any associated external streams.

```
virtual int sync();
```

Return Value

If the function cannot succeed, it returns -1. The default behavior is to return zero.

Remarks

`sync` involves writing out any elements between the beginning and next pointers for the output buffer. It does not involve putting back any elements between the next and end pointers for the input buffer.

basic_streambuf::traits_type

Associates a type name with the **Tr** template parameter.

```
typedef Tr traits_type;
```

basic_streambuf::uflow

A protected virtual function that extracts the current element from the input stream.

```
virtual int_type uflow();
```

Return Value

The current element.

Remarks

The protected virtual member function tries to extract the current element **ch** from the input stream, then advance the current stream position, and return the element as **traits_type::to_int_type(ch)**. It can do so in various ways:

- If a read position is available, it takes **ch** as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value **ch**.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function cannot succeed, it returns **traits_type::eof**, or throws an exception. Otherwise, it returns the current element `ch` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call `underflow` and, if that function returns **traits_type::eof**, to return **traits_type::eof**. Otherwise, the function returns the current element **ch** in the input stream, converted as previously described, and advances the next pointer for the input buffer.

basic_streambuf::underflow

Protected, virtual function to extract the current element from the input stream.

```
virtual int_type underflow();
```

Return Value

The current element.

Remarks

The protected virtual member function endeavors to extract the current element **ch** from the input stream, without advancing the current stream position, and return it as `traits_type::to_int_type(ch)`. It can do so in

various ways:

- If a read position is available, **ch** is the element stored in the read position. For more information on this, see the Remarks section of the [basic_streambuf Class](#).
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements. For more information on this, see the Remarks section of the [basic_streambuf Class](#).

If the function cannot succeed, it returns `traits_type::eof()` or throws an exception. Otherwise, it returns the current element in the input stream, converted as previously described. The default behavior is to return `traits_type::eof()`.

The virtual `underflow` function, with the [sync](#) and [overflow](#) functions, defines the characteristics of the `streambuf`-derived class. Each derived class might implement `underflow` differently, but the interface with the calling stream class is the same.

The `underflow` function is most frequently called by public `streambuf` functions like [sgetc](#) and [sgetn](#) when the get area is empty, but other classes, including the stream classes, can call `underflow` anytime.

The `underflow` function supplies the get area with characters from the input source. If the get area contains characters, `underflow` returns the first character. If the get area is empty, it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then `underflow` returns `EOF` and leaves the get area empty.

In the `strstreambuf` class, `underflow` adjusts the [egptr](#) pointer to access storage that was dynamically allocated by a call to `overflow`.

basic_streambuf::xsgetn

Protected, virtual function to extract elements from the input stream.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
virtual streamsize xsgetn(  
    char_type* ptr,  
    streamsize count);
```

Parameters

ptr

The buffer to contain the extracted characters.

count

The number of elements to extract.

Return Value

The number of elements extracted.

Remarks

The protected virtual member function extracts up to *count* elements from the input stream, as if by repeated calls to [sbumpc](#), and stores them in the array beginning at *ptr*. It returns the number of elements actually extracted.

basic_streambuf::xsputn

Protected, virtual function to insert elements into the output stream.

```
virtual streamsize xspn(const char_type* ptr, streamsize count);
```

Parameters

ptr

Pointer to elements to insert.

count

Number of elements to insert.

Return Value

The number of elements actually inserted into the stream.

Remarks

The protected virtual member function inserts up to *count* elements into the output stream, as if by repeated calls to [sputc](#), from the array beginning at *ptr*. The insertion of characters into the output stream stops once all *count* characters have been written, or if calling `sputc(count)` would return `traits::eof()`. It returns the number of elements actually inserted.

See also

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<string>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class `basic_string` and various supporting templates.

For more information about `basic_string`, see [basic_string Class](#)

Syntax

```
#include <string>
```

Remarks

The C++ language and the C++ Standard Library support two types of strings:

- Null-terminated character arrays often referred to as C strings.
- Template class objects, of type `basic_string`, that handle all **char**-like template arguments.

Typedefs

TYPE NAME	DESCRIPTION
<code>string</code>	A type that describes a specialization of the template class <code>basic_string</code> with elements of type char as a <code>string</code> .
<code>wstring</code>	A type that describes a specialization of the template class <code>basic_string</code> with elements of type wchar_t as a <code>wstring</code> .
<code>u16string</code>	A type that describes a specialization of the template class <code>basic_string</code> based on elements of type <code>char16_t</code> .
<code>u32string</code>	A type that describes a specialization of the template class <code>basic_string</code> based on elements of type <code>char32_t</code> .

Operators

OPERATOR	DESCRIPTION
<code>operator+</code>	Concatenates two string objects.
<code>operator!=</code>	Tests if the string object on the left side of the operator is not equal to the string object on the right side.
<code>operator==</code>	Tests if the string object on the left side of the operator is equal to the string object on the right side.
<code>operator<</code>	Tests if the string object on the left side of the operator is less than to the string object on the right side.

OPERATOR	DESCRIPTION
<code>operator<=</code>	Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.
<code>operator<<</code>	A template function that inserts a string into the output stream.
<code>operator></code>	Tests if the string object on the left side of the operator is greater than to the string object on the right side.
<code>operator>=</code>	Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.
<code>operator>></code>	A template function that extracts a string from the input stream.

Specialized Template Functions

<code>swap</code>	Exchanges the arrays of characters of two strings.
<code>stod</code>	Converts a character sequence to a double .
<code>stof</code>	Converts a character sequence to a float .
<code>stoi</code>	Converts a character sequence to an integer.
<code>stold</code>	Converts a character sequence to a long double .
<code>stoll</code>	Converts a character sequence to a long long .
<code>stoul</code>	Converts a character sequence to an unsigned long .
<code>stoull</code>	Converts a character sequence to an unsigned long long .
<code>to_string</code>	Converts a value to a <code>string</code> .
<code>to_wstring</code>	Converts a value to a wide <code>string</code> .

Functions

FUNCTION	DESCRIPTION
<code>getline Template</code>	Extract strings from the input stream line by line.

Classes

CLASS	DESCRIPTION
<code>basic_string Class</code>	A template class that describes objects that can store a sequence of arbitrary character-like objects.

CLASS	DESCRIPTION
char_traits Struct	A template class that describes attributes associated with a character of type CharType

Specializations

char_traits<char> Struct	A struct that is a specialization of the template struct <code>char_traits<CharType></code> to an element of type <code>char</code> .
char_traits<wchar_t> Struct	A struct that is a specialization of the template struct <code>char_traits<CharType></code> to an element of type <code>wchar_t</code> .
char_traits<char16_t> Struct	A struct that is a specialization of the template struct <code>char_traits<CharType></code> to an element of type <code>char16_t</code> .
char_traits<char32_t> Struct	A struct that is a specialization of the template struct <code>char_traits<CharType></code> to an element of type <code>char32_t</code> .

Requirements

- **Header:** `<string>`
- **Namespace:** `std`

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<string> functions

11/9/2018 • 11 minutes to read • [Edit Online](#)

getline	stod	stof
stoi	stol	stold
stoll	stoul	stoull
swap	to_string	to_wstring

getline

Extract strings from the input stream line-by-line.

```
// (1) delimiter as parameter
template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>& is,
    basic_string<CharType, Traits, Allocator>& str,
    CharType delim);

template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>&& is,
    basic_string<CharType, Traits, Allocator>& str,
    const CharType delim);

// (2) default delimiter used
template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>& is,
    basic_string<CharType, Traits, Allocator>& str);

template <class Allocator, class Traits, class Allocator>
basic_istream<Allocator, Traits>& getline(
    basic_istream<Allocator, Traits>&& is,
    basic_string<Allocator, Traits, Allocator>& str);
```

Parameters

is

The input stream from which a string is to be extracted.

str

The string into which are read the characters from the input stream.

delim

The line delimiter.

Return Value

The input stream *is*.

Remarks

The pair of function signatures marked (1) extract characters from *is* until *delim* is found, storing them in *str*.

The pair of function signatures marked (2) use newline as the default line delimiter and behave as `getline(is, str, is.widen('\n'))`.

The second function of each pair is an analog to the first one to support [rvalue references](#).

Extraction stops when one of the following occurs:

- At end-of-file, in which case the internal state flag of *is* is set to `ios_base::eofbit`.
- After the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence.
- After the function extracts `str.max_size` elements, in which case the internal state flag of *is* is set to `ios_base::failbit`.
- Some other error other than those previously listed, in which case the internal state flag of *is* is set to `ios_base::badbit`.

For information about internal state flags, see [ios_base::iostate](#).

If the function extracts no elements, the internal state flag of *is* is set to `ios_base::failbit`. In any case, `getline` returns *is*.

If an exception is thrown, *is* and *str* are left in a valid state.

Example

The following code demonstrates `getline()` in two modes: first with the default delimiter (newline) and second with a whitespace as delimiter. The end-of-file character (CTRL-Z on the keyboard) is used to control termination of the while loops. This sets the internal state flag of `cin` to `eofbit`, which must be cleared with [basic_ios::clear\(\)](#) before the second while loop will work properly.

```

// compile with: /EHsc /W4
#include <string>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    string str;
    vector<string> v1;
    cout << "Enter a sentence, press ENTER between sentences. (Ctrl-Z to stop): " << endl;
    // Loop until end-of-file (Ctrl-Z) is input, store each sentence in a vector.
    // Default delimiter is the newline character.
    while (getline(cin, str)) {
        v1.push_back(str);
    }

    cout << "The following input was stored with newline delimiter:" << endl;
    for (const auto& p : v1) {
        cout << p << endl;
    }

    cin.clear();

    vector<string> v2;
    // Now try it with a whitespace delimiter
    while (getline(cin, str, ' ')) {
        v2.push_back(str);
    }

    cout << "The following input was stored with whitespace as delimiter:" << endl;
    for (const auto& p : v2) {
        cout << p << endl;
    }
}

```

stod

Converts a character sequence to a **double**.

```

double stod(
    const string& str,
    size_t* idx = 0);

double stod(
    const wstring& str,
    size_t* idx = 0
);

```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.

Return Value

The **double** value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **double** as if by calling `strtod(str.c_str(), _Eptr)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stof

Converts a character sequence to a float.

```
float stof(
    const string& str,
    size_t* idx = 0);

float stof(
    const wstring& str,
    size_t* idx = 0);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.

Return Value

The float value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **float** as if by calling `strtof(str.c_str(), _Eptr)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stoi

Converts a character sequence to an integer.

```
int stoi(
    const string& str,
    size_t* idx = 0,
    int base = 10);

int stoi(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Return Value

The integer value.

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	Contains the index of the first unconverted character on return.
<i>base</i>	The number base to use.

Remarks

The function `stoi` converts the sequence of characters in *str* to a value of type **int** and returns the value. For example, when passed a character sequence "10", the value returned by `stoi` is the integer 10.

`stoi` behaves similarly to the function `strtol` for single-byte characters when it is called in the manner `strtol(str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function; or `wcstol` for wide characters, when it is called in similar manner, `wcstol(Str.c_str(), _Eptr, idx)`. For more information, see [strtol](#), [wcstol](#), [_strtol_l](#), [_wcstol_l](#).

If `str.c_str() == *_Eptr`, `stoi` throws an object of type `invalid_argument`. If such a call would set `errno`, or if the returned value cannot be represented as an object of type **int**, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx`.

stol

Converts a character sequence to a **long**.

```
long stol(
    const string& str,
    size_t* idx = 0,
    int base = 10);

long stol(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.
<i>base</i>	The number base to use.

Return Value

The long-integer value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **long** as if by calling `strtol(str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stold

Converts a character sequence to a **long double**.

```
double stold(
    const string& str,
    size_t* idx = 0);

double stold(
    const wstring& str,
    size_t* idx = 0);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.

Return Value

The **long double** value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **long double** as if by calling `strtold(str.c_str(), _Eptr)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stoll

Converts a character sequence to a **long long**.

```
long long stoll(
    const string& str,
    size_t* idx = 0,
    int base = 10);

long long stoll(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.
<i>base</i>	The number base to use.

Return Value

The **long long** value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **long long** as if by calling `strtoll(str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stoul

Converts a character sequence to an unsigned long.

```
unsigned long stoul(
    const string& str,
    size_t* idx = 0,
    int base = 10);

unsigned long stoul(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.
<i>base</i>	The number base to use.

Return Value

The unsigned long-integer value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **unsigned long** as if by calling `strtoul(str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

stoull

Converts a character sequence to an **unsigned long long**.

```
unsigned long long stoull(
    const string& str,
    size_t* idx = 0,
    int base = 10);

unsigned long long stoull(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

PARAMETER	DESCRIPTION
<i>str</i>	The character sequence to be converted.
<i>idx</i>	The index value of the first unconverted character.
<i>base</i>	The number base to use.

Return Value

The **unsigned long long** value.

Remarks

The function converts the sequence of elements in *str* to a value `val` of type **unsigned long long** as if by calling `strtoull(str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr` it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* is not a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns `val`.

swap

Exchanges the arrays of characters of two strings.

```
template <class Traits, class Allocator>
void swap(basic_string<CharType, Traits, Allocator>& left, basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

One string whose elements are to be swapped with those of another string.

right

The other string whose elements are to be swapped with the first string.

Remarks

The template function executes the specialized member function *left.swap(right)* for strings, which guarantees constant complexity.

Example

```
// string_swap.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    // Declaring an object of type basic_string<char>
    string s1 ( "Tweedledee" );
    string s2 ( "Tweedledum" );
    cout << "Before swapping string s1 and s2:" << endl;
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    swap ( s1 , s2 );
    cout << "\nAfter swapping string s1 and s2:" << endl;
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;
}
```

Before swapping string s1 and s2:
The basic_string s1 = Tweedledee.
The basic_string s2 = Tweedledum.

After swapping string s1 and s2:
The basic_string s1 = Tweedledum.
The basic_string s2 = Tweedledee.

to_string

Converts a value to a `string`.

```
string to_string(int Val);
string to_string(unsigned int Val);
string to_string(long Val);
string to_string(unsigned long Val);
string to_string(long long Val);
string to_string(unsigned long long Val);
string to_string(float Val);
string to_string(double Val);
string to_string(long double Val);
```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value to be converted.

Return Value

The `string` that represents the value.

Remarks

The function converts *Val* to a sequence of elements stored in an array object `Buf` internal to the function as if by calling `sprintf(Buf, Fmt, Val)`, where `Fmt` is

- `"%d"` if `Val` has type **int**

- "%u" if Val has type **unsigned int**
- "%ld" if Val has type **long**
- "%lu" if Val has type **unsigned long**
- "%lld" if Val has type **long long**
- "%llu" if Val has type **unsigned long long**
- "%f" if Val has type **float** or **double**
- "%Lf" if Val has type **long double**

The function returns `string(Buf)` .

to_wstring

Converts a value to a wide string.

```
wstring to_wstring(int Val);
wstring to_wstring(unsigned int Val);
wstring to_wstring(long Val);
wstring to_wstring(unsigned long Val);
wstring to_wstring(long long Val);
wstring to_wstring(unsigned long long Val);
wstring to_wstring(float Val);
wstring to_wstring(double Val);
wstring to_wstring(long double Val);
```

Parameters

PARAMETER	DESCRIPTION
Val	The value to be converted.

Return Value

The wide string that represents the value.

Remarks

The function converts Val to a sequence of elements stored in an array object Buf internal to the function as if by calling `swprintf(Buf, Len, Fmt, Val)` , where Fmt is

- L"%d" if Val has type **int**
- L"%u" if Val has type **unsigned int**
- L"%ld" if Val has type **long**
- L"%lu" if Val has type **unsigned long**
- L"%lld" if Val has type **long long**
- L"%llu" if Val has type **unsigned long long**
- L"%f" if Val has type **float** or **double**
- L"%Lf" if Val has type **long double**

The function returns `wstring(Buf)` .

See also

[`<string>`](#)

<string> operators

11/8/2018 • 18 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>></code>
<code>operator>=</code>	<code>operator<</code>	<code>operator<<</code>
<code>operator<=</code>	<code>operator+</code>	<code>operator==</code>

operator+

Concatenates two string objects.

```

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const CharType left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>&& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    CharType right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>&& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    CharType left,
    const basic_string<CharType, Traits, Allocator>&& right);

```

Parameters

left

A C-style string or an object of type `basic_string` to be concatenated.

right

A C-style string or an object of type `basic_string` to be concatenated.

Return Value

The string that is the concatenation of the input strings.

Remarks

The functions each overload `operator+` to concatenate two objects of template class `basic_string Class`. All effectively return `basic_string< CharType, Traits, Allocator>(Left).append(right)`. For more information, see [append](#).

Example

```
// string_op_con.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    // Declaring an object of type basic_string<char>
    string s1 ( "anti" );
    string s2 ( "gravity" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "heroine";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // Declaring a character constant
    char c1 = '!';
    cout << "The character constant c1 = " << c1 << "." << endl;

    // First member function: concatenates an object
    // of type basic_string with an object of type basic_string
    string s12 = s1 + s2;
    cout << "The string concatenating s1 & s2 is: " << s12 << endl;

    // Second & fourth member functions: concatenate an object
    // of type basic_string with an object of C-style string type
    string s1s3 = s1 + s3;
    cout << "The string concatenating s1 & s3 is: " << s1s3 << endl;

    // Third & fifth member functions: concatenate an object
    // of type basic_string with a character constant
    string s1s3c1 = s1s3 + c1;
    cout << "The string concatenating s1 & s3 is: " << s1s3c1 << endl;
}
```

```
The basic_string s1 = anti.
The basic_string s2 = gravity.
The C-style string s3 = heroine.
The character constant c1 = !.
The string concatenating s1 & s2 is: antigravity
The string concatenating s1 & s3 is: antiheroine
The string concatenating s1 & s3 is: antiheroine!
```

operator!=

Tests if the string object on the left side of the operator is not equal to the string object on the right side.


```
template <class CharType, class Traits, class Allocator>
bool operator!=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator!=(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator!=(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is not lexicographically equal to the string object on the right side; otherwise **false**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of characters and their respective character values are the same. Otherwise, they are unequal.

Example

```

// string_op_ne.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "pluck" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "pluck";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 != s2 )
        cout << "The strings s1 & s2 are not equal." << endl;
    else
        cout << "The strings s1 & s2 are equal." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s1 != s3 )
        cout << "The strings s1 & s3 are not equal." << endl;
    else
        cout << "The strings s1 & s3 are equal." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s3 != s2 )
        cout << "The strings s3 & s2 are not equal." << endl;
    else
        cout << "The strings s3 & s2 are equal." << endl;
}

```

```

The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.

```

operator==

Tests if the string object on the left side of the operator is equal to the string object on the right side.

```
template <class CharType, class Traits, class Allocator>
bool operator==(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator==(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator==(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically equal to the string object on the right side; otherwise **false**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of characters and their respective character values are the same. Otherwise, they are unequal.

Example

```

// string_op_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "pluck" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "pluck";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 == s2 )
        cout << "The strings s1 & s2 are equal." << endl;
    else
        cout << "The strings s1 & s2 are not equal." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s1 == s3 )
        cout << "The strings s1 & s3 are equal." << endl;
    else
        cout << "The strings s1 & s3 are not equal." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s3 == s2 )
        cout << "The strings s3 & s2 are equal." << endl;
    else
        cout << "The strings s3 & s2 are not equal." << endl;
}

```

```

The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.

```

operator<

Tests if the string object on the left side of the operator is less than to the string object on the right side.

```

template <class CharType, class Traits, class Allocator>
bool operator<(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator<(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator<(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);

```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically less than the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

```

// string_op_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    // Declaring an objects of type basic_string<char>
    string s1 ( "strict" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "strict";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 < s2 )
        cout << "The string s1 is less than the string s2." << endl;
    else
        cout << "The string s1 is not less than the string s2." << endl;

    // Second member function: comparison between left-hand object
    // of type basic_string & right-hand object of C-syle string type
    if ( s1 < s3 )
        cout << "The string s1 is less than the string s3." << endl;
    else
        cout << "The string s1 is not less than the string s3." << endl;

    // Third member function: comparison between left-hand object
    // of C-syle string type & right-hand object of type basic_string
    if ( s3 < s2 )
        cout << "The string s3 is less than the string s2." << endl;
    else
        cout << "The string s3 is not less than the string s2." << endl;
}

```

```

The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than the string s2.
The string s1 is not less than the string s3.
The string s3 is less than the string s2.

```

operator<=

Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.

```
template <class CharType, class Traits, class Allocator>
bool operator<=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator<=(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator<=(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically less than or equal to the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

```

// string_op_le.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "strict" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "strict";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 <= s2 )
        cout << "The string s1 is less than or equal to "
            << "the string s2." << endl;
    else
        cout << "The string s1 is greater than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s1 <= s3 )
        cout << "The string s1 is less than or equal to "
            << "the string s3." << endl;
    else
        cout << "The string s1 is greater than "
            << "the string s3." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s2 <= s3 )
        cout << "The string s2 is less than or equal to "
            << "the string s3." << endl;
    else
        cout << "The string s2 is greater than "
            << "the string s3." << endl;
}

```

```

The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than or equal to the string s2.
The string s1 is less than or equal to the string s3.
The string s2 is greater than the string s3.

```

operator<<

A template function that writes a string into the output stream.

```

template <class CharType, class Traits, class Allocator>
basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& _Ostr,
    const basic_string<CharType, Traits, Allocator>& str);

```


Parameters

_Ostr

The output stream being written to.

str

The string to be entered into the output stream.

Return Value

Writes the value of the specified string to the output stream *_Ostr*.

Remarks

The template function overloads **operator<<** to insert an object *str* of template class [basic_string](#) into the stream *_Ostr*. The function effectively returns `_Ostr.write(str.c_str, str.size)`.

operator>

Tests if the string object on the left side of the operator is greater than to the string object on the right side.

```
template <class CharType, class Traits, class Allocator>
bool operator>(  
    const basic_string<CharType, Traits, Allocator>& left,  
    const basic_string<CharType, Traits, Allocator>& right);  
  
template <class CharType, class Traits, class Allocator>  
bool operator>(  
    const basic_string<CharType, Traits, Allocator>& left,  
    const CharType* right);  
  
template <class CharType, class Traits, class Allocator>  
bool operator>(  
    const CharType* left,  
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically greater than the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

```

// string_op_gt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "strict" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "stricture";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 > s2 )
        cout << "The string s1 is greater than "
            << "the string s2." << endl;
    else
        cout << "The string s1 is not greater than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s3 > s1 )
        cout << "The string s3 is greater than "
            << "the string s1." << endl;
    else
        cout << "The string s3 is not greater than "
            << "the string s1." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s2 > s3 )
        cout << "The string s2 is greater than "
            << "the string s3." << endl;
    else
        cout << "The string s2 is not greater than "
            << "the string s3." << endl;
}

```

```

The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is not greater than the string s2.
The string s3 is greater than the string s1.
The string s2 is greater than the string s3.

```

operator>=

Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.

```

template <class CharType, class Traits, class Allocator>
bool operator>=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator>=(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator>=(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);

```

Parameters

left

A C-style string or an object of type `basic_string` to be compared.

right

A C-style string or an object of type `basic_string` to be compared.

Return Value

true if the string object on the left side of the operator is lexicographically greater than or equal to the string object on the right side; otherwise **false**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds the strings have the same number of characters, and so the strings are equal.

Example

```

// string_op_ge.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "strict" );
    string s2 ( "strum" );
    cout << "The basic_string s1 = " << s1 << "." << endl;
    cout << "The basic_string s2 = " << s2 << "." << endl;

    // Declaring a C-style string
    char *s3 = "stricture";
    cout << "The C-style string s3 = " << s3 << "." << endl;

    // First member function: comparison between left-side object
    // of type basic_string & right-side object of type basic_string
    if ( s1 >= s2 )
        cout << "The string s1 is greater than or equal to "
            << "the string s2." << endl;
    else
        cout << "The string s1 is less than "
            << "the string s2." << endl;

    // Second member function: comparison between left-side object
    // of type basic_string & right-side object of C-syle string type
    if ( s3 >= s1 )
        cout << "The string s3 is greater than or equal to "
            << "the string s1." << endl;
    else
        cout << "The string s3 is less than "
            << "the string s1." << endl;

    // Third member function: comparison between left-side object
    // of C-syle string type & right-side object of type basic_string
    if ( s2 >= s3 )
        cout << "The string s2 is greater than or equal to "
            << "the string s3." << endl;
    else
        cout << "The string s2 is less than "
            << "the string s3." << endl;
}

```

```

The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is less than the string s2.
The string s3 is greater than or equal to the string s1.
The string s2 is greater than or equal to the string s3.

```

operator>>

A template function that reads a string from an input stream.

```

template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& operator>>(
    basic_istream<CharType, Traits>& _Istr,
    basic_string<CharType, Traits, Allocator>& right);

```

Parameters

_Istr

The input stream used to extract the sequence

right

The string that is being extracted from the input stream.

Return Value

Reads the value of the specified string from *_Istr* and returns it into *right*.

Remarks

The operator skips the leading white spaces unless the `skipws` flag is set. It reads all the following characters until the next character is a white space or the end of the file is reached.

The template function overloads **operator>>** to replace the sequence controlled by *right* with a sequence of elements extracted from the stream *_Istr*. Extraction stops:

- At end of file.
- After the function extracts `_Istr.width` elements, if that value is nonzero.

After the function extracts `_Istr.max_size` elements.

- After the function extracts an element *ch* for which `use_facet<ctype< CharType>> (> (getloc)).is(ctype< CharType>::space, ch)` is true, in which case the character is put back.

If the function extracts no elements, it calls `setstate(ios_base::failbit)`. In any case, it calls `istr.width(0)` and returns `* this`.

Example

```
// string_op_read_.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    string c0;
    cout << "Input a string c0 ( try: Fibonacci numbers ): ";
    cin >> c0;
    cout << "The string entered is c0 = " << c0 << endl;
}
```

See also

[<string>](#)

<string> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

string	u16string	u32string
wstring		

string

A type that describes a specialization of the template class [basic_string](#) with elements of type **char**.

Other typedefs that specialize `basic_string` include [wstring](#), [u16string](#), and [u32string](#).

```
typedef basic_string<char, char_traits<char>, allocator<char>> string;
```

Remarks

The following are equivalent declarations:

```
string str("");  
  
basic_string<char> str("");
```

For a list of string constructors, see [basic_string::basic_string](#).

u16string

A type that describes a specialization of the template class [basic_string](#) with elements of type `char16_t`.

Other typedefs that specialize `basic_string` include [wstring](#), [string](#), and [u32string](#).

```
typedef basic_string<char16_t, char_traits<char16_t>, allocator<char16_t>> u16string;
```

Remarks

For a list of string constructors, see [basic_string::basic_string](#).

u32string

A type that describes a specialization of the template class [basic_string](#) with elements of type `char32_t`.

Other typedefs that specialize `basic_string` include [string](#), [u16string](#), and [wstring](#).

```
typedef basic_string<char32_t, char_traits<char32_t>, allocator<char32_t>> u32string;
```

Remarks

For a list of string constructors, see [basic_string::basic_string](#).

wstring

A type that describes a specialization of the template class [basic_string](#) with elements of type **wchar_t**.

Other typedefs that specialize `basic_string` include [string](#), [u16string](#), and [u32string](#).

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>> wstring;
```

Remarks

The following are equivalent declarations:

```
wstring wstr(L "");  
  
basic_string<wchar_t> wstr(L "");
```

For a list of string constructors, see [basic_string::basic_string](#).

NOTE

The size of **wchar_t** is implementation-defined. If your code depends on **wchar_t** to be a certain size, check your platform's implementation (for example, with `sizeof(wchar_t)`). If you need a string character type with a width that is guaranteed to remain the same on all platforms, use [string](#), [u16string](#), or [u32string](#).

See also

[<string>](#)

basic_string Class

10/31/2018 • 108 minutes to read • [Edit Online](#)

The sequences controlled by an object of template class `basic_string` are the Standard C++ string class and are usually referred to as strings, but they should not be confused with the null-terminated C-style strings used throughout the C++ Standard Library. The Standard C++ string is a container that enables the use of strings as normal types, such as comparison and concatenation operations, iterators, C++ Standard Library algorithms, and copying and assigning with class allocator managed memory. If you need to convert a Standard C++ string to a null-terminated C-style string, use the `basic_string::c_str` member.

Syntax

```
template <class CharType, class Traits = char_traits<CharType>, class Allocator = allocator<CharType>>
class basic_string;
```

Parameters

CharType

The data type of a single character to be stored in the string. The C++ Standard Library provides specializations of this template class, with the type definitions `string` for elements of type **char**, `wstring`, for **wchar_t**, `u16string` for `char16_t`, and `u32string` for `char32_t`.

Traits

Various important properties of the `CharType` elements in a `basic_string` specialization are described by the class `Traits`. The default value is `char_traits < CharType >`.

Allocator

The type that represents the stored allocator object that encapsulates details about the string's allocation and deallocation of memory. The default value is `allocator< CharType >`.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>basic_string</code>	Constructs a string that is empty or initialized by specific characters or that is a copy of all or part of some other string object or C-string.

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for a string object.
<code>const_iterator</code>	A type that provides a random-access iterator that can access and read a const element in the string.
<code>const_pointer</code>	A type that provides a pointer to a const element in a string.

TYPE NAME	DESCRIPTION
<code>const_reference</code>	A type that provides a reference to a const element stored in a string for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a random-access iterator that can read any const element in the string.
<code>difference_type</code>	A type that provides the difference between two iterators that refer to elements within the same string.
<code>iterator</code>	A type that provides a random-access iterator that can read or modify any element in a string.
<code>npos</code>	An unsigned integral value initialized to -1 that indicates either "not found" or "all remaining characters" when a search function fails.
<code>pointer</code>	A type that provides a pointer to a character element in a string or character array.
<code>reference</code>	A type that provides a reference to an element stored in a string.
<code>reverse_iterator</code>	A type that provides a random-access iterator that can read or modify an element in a reversed string.
<code>size_type</code>	An unsigned integral type for the number of elements in a string.
<code>traits_type</code>	A type for the character traits of the elements stored in a string.
<code>value_type</code>	A type that represents the type of characters stored in a string.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>append</code>	Adds characters to the end of a string.
<code>assign</code>	Assigns new character values to the contents of a string.
<code>at</code>	Returns a reference to the element at a specified location in the string.
<code>back</code>	
<code>begin</code>	Returns an iterator addressing the first element in the string.
<code>c_str</code>	Converts the contents of a string as a C-style, null-terminated, string.

MEMBER FUNCTION	DESCRIPTION
capacity	Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.
cbegin	Returns a const iterator addressing the first element in the string.
cend	Returns a const iterator that addresses the location succeeding the last element in a string.
clear	Erases all elements of a string.
compare	Compares a string with a specified string to determine if the two strings are equal or if one is lexicographically less than the other.
copy	Copies at most a specified number of characters from an indexed position in a source string to a target character array. Deprecated. Use basic_string::_Copy_s instead.
crbegin	Returns a const iterator that addresses the first element in a reversed string.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed string.
_Copy_s	Copies at most a specified number of characters from an indexed position in a source string to a target character array.
data	Converts the contents of a string into an array of characters.
empty	Tests whether the string contains characters.
end	Returns an iterator that addresses the location succeeding the last element in a string.
erase	Removes an element or a range of elements in a string from a specified position.
find	Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.
find_first_not_of	Searches through a string for the first character that is not any element of a specified string.
find_first_of	Searches through a string for the first character that matches any element of a specified string.
find_last_not_of	Searches through a string for the last character that is not any element of a specified string.

MEMBER FUNCTION	DESCRIPTION
<code>find_last_of</code>	Searches through a string for the last character that is an element of a specified string.
<code>front</code>	Returns a reference to the first element in a string.
<code>get_allocator</code>	Returns a copy of the <code>allocator</code> object used to construct the string.
<code>insert</code>	Inserts an element or a number of elements or a range of elements into the string at a specified position.
<code>length</code>	Returns the current number of elements in a string.
<code>max_size</code>	Returns the maximum number of characters a string could contain.
<code>pop_back</code>	Erases the last element of the string.
<code>push_back</code>	Adds an element to the end of the string.
<code>rbegin</code>	Returns an iterator to the first element in a reversed string.
<code>rend</code>	Returns an iterator that points just beyond the last element in a reversed string.
<code>replace</code>	Replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.
<code>reserve</code>	Sets the capacity of the string to a number at least as great as a specified number.
<code>resize</code>	Specifies a new size for a string, appending or erasing elements as required.
<code>rfind</code>	Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.
<code>shrink_to_fit</code>	Discards the excess capacity of the string.
<code>size</code>	Returns the current number of elements in a string.
<code>substr</code>	Copies a substring of at most some number of characters from a string beginning from a specified position.
<code>swap</code>	Exchange the contents of two strings.

Operators

OPERATOR	DESCRIPTION
----------	-------------

OPERATOR	DESCRIPTION
<code>operator+=</code>	Appends characters to a string.
<code>operator=</code>	Assigns new character values to the contents of a string.
<code>operator[]</code>	Provides a reference to the character with a specified index in a string.

Remarks

If a function is asked to generate a sequence longer than `max_size` elements, the function reports a length error by throwing an object of type `length_error`.

References, pointers, and iterators that designate elements of the controlled sequence can become invalid after any call to a function that alters the controlled sequence, or after the first call to a non- **const** member function.

Requirements

Header: `<string>`

Namespace: `std`

`basic_string::allocator_type`

A type that represents the allocator class for a string object.

```
typedef Allocator allocator_type;
```

Remarks

The type is a synonym for the template parameter `Allocator`.

Example

```
// basic_string_allocator_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    // The following lines declare objects
    // that use the default allocator.
    string s1;
    basic_string<char>::allocator_type xchar = s1.get_allocator( );
    // You can now call functions on the allocator class xchar used by s1
}
```

`basic_string::append`

Adds characters to the end of a string.

```

basic_string<CharType, Traits, Allocator>& append(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& append(
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& append(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off,
    size_type count);

basic_string<CharType, Traits, Allocator>& append(
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& append(
    size_type count,
    value_type _Ch);

template <class InputIterator>
basic_string<CharType, Traits, Allocator>& append(
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& append(
    const_pointer first,
    const_pointer last);

basic_string<CharType, Traits, Allocator>& append(
    const_iterator first,
    const_iterator last);

```

Parameters

ptr

The C-string to be appended.

str

The string whose characters are to be appended.

_Off

The index of the part of the source string supplying the characters to be appended.

count

The number of characters to be appended, at most, from the source string.

_Ch

The character value to be appended.

first

An input iterator addressing the first element in the range to be appended.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the position of the one beyond the last element in the range to be appended.

Return Value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the [operator+=](#) or the member functions `append` or `push_back`. `operator+=` appends single-argument values while the multiple-argument `append` member

function allows a specific part of a string to be specified for adding.

Example

```

// basic_string_append.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // appending a C-string to a string
    string str1a ( "Hello " );
    cout << "The original string str1 is: " << str1a << endl;
    const char *cstr1a = "Out There ";
    cout << "The C-string cstr1a is: " << cstr1a << endl;
    str1a.append ( cstr1a );
    cout << "Appending the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

    // The second member function
    // appending part of a C-string to a string
    string str1b ( "Hello " );
    cout << "The string str1b is: " << str1b << endl;
    const char *cstr1b = "Out There ";
    cout << "The C-string cstr1b is: " << cstr1b << endl;
    str1b.append ( cstr1b , 3 );
    cout << "Appending the 1st part of the C-string cstr1b "
        << "to string str1 gives: " << str1b << "."
        << endl << endl;

    // The third member function
    // appending part of one string to another
    string str1c ( "Hello " ), str2c ( "Wide World " );
    cout << "The string str2c is: " << str2c << endl;
    str1c.append ( str2c , 5 , 5 );
    cout << "The appended string str1 is: "
        << str1c << "." << endl << endl;

    // The fourth member function
    // appending one string to another in two ways,
    // comparing append and operator [ ]
    string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World " );
    cout << "The string str2d is: " << str2d << endl;
    str1d.append ( str2d );
    cout << "The appended string str1d is: "
        << str1d << "." << endl;
    str1d += str3d;
    cout << "The doubly appended string str1 is: "
        << str1d << "." << endl << endl;

    // The fifth member function
    // appending characters to a string
    string str1e ( "Hello " );
    str1e.append ( 4 , '!' );
    cout << "The string str1 appended with exclamations is: "
        << str1e << endl << endl;

    // The sixth member function
    // appending a range of one string to another
    string str1f ( "Hello " ), str2f ( "Wide World " );
    cout << "The string str2f is: " << str2f << endl;
    str1f.append ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
    cout << "The appended string str1 is: "
        << str1f << "." << endl << endl;
}

```

```
The original string str1 is: Hello
The C-string cstr1a is: Out There
Appending the C-string cstr1a to string str1 gives: Hello Out There .

The string str1b is: Hello
The C-string cstr1b is: Out There
Appending the 1st part of the C-string cstr1b to string str1 gives: Hello Out.

The string str2c is: Wide World
The appended string str1 is: Hello World.

The string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended strig str1 is: Hello Wide World .

The string str1 appended with exclamations is: Hello !!!!

The string str2f is: Wide World
The appended string str1 is: Hello World.
```

basic_string::assign

Assigns new character values to the contents of a string.

```
basic_string<CharType, Traits, Allocator>& assign(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& assign(
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& assign(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type off,
    size_type count);

basic_string<CharType, Traits, Allocator>& assign(
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& assign(
    size_type count,
    value_type _Ch);

template <class InIt>
basic_string<CharType, Traits, Allocator>& assign(
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& assign(
    const_pointer first,
    const_pointer last);

basic_string<CharType, Traits, Allocator>& assign(
    const_iterator first,
    const_iterator last);
```

Parameters

ptr

A pointer to the characters of the C-string to be assigned to the target string.

count

The number of characters to be assigned, from the source string.

str

The source string whose characters are to be assigned to the target string.

_Ch

The character value to be assigned.

first

An input iterator, `const_pointer`, or `const_iterator` addressing the first character in the range of the source string to be assigned to the target range.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the one beyond the last character in the range of the source string to be assigned to the target range.

off

The position at which new characters will start to be assigned.

Return Value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings can be assigned new character values. The new value can be either a string and C-string or a single character. The `operator=` may be used if the new value can be described by a single parameter; otherwise the member function `assign`, which has multiple parameters, can be used to specify which part of the string is to be assigned to a target string.

Example

```

// basic_string_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function assigning the
    // characters of a C-string to a string
    string str1a;
    const char *cstr1a = "Out There";
    cout << "The C-string cstr1a is: " << cstr1a << "." << endl;
    str1a.assign ( cstr1a );
    cout << "Assigning the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

    // The second member function assigning a specific
    // number of the of characters a C-string to a string
    string str1b;
    const char *cstr1b = "Out There";
    cout << "The C-string cstr1b is: " << cstr1b << endl;
    str1b.assign ( cstr1b , 3 );
    cout << "Assigning the 1st part of the C-string cstr1b "
        << "to string str1 gives: " << str1b << "."
        << endl << endl;

    // The third member function assigning a specific number
    // of the characters from one string to another string
    string str1c ( "Hello " ), str2c ( "Wide World " );
    cout << "The string str2c is: " << str2c << endl;
    str1c.assign ( str2c , 5 , 5 );
    cout << "The newly assigned string str1 is: "
        << str1c << "." << endl << endl;

    // The fourth member function assigning the characters
    // from one string to another string in two equivalent
    // ways, comparing the assign and operator =
    string str1d ( "Hello" ), str2d ( "Wide" ), str3d ( "World" );
    cout << "The original string str1 is: " << str1d << "." << endl;
    cout << "The string str2d is: " << str2d << endl;
    str1d.assign ( str2d );
    cout << "The string str1 newly assigned with string str2d is: "
        << str1d << "." << endl;
    cout << "The string str3d is: " << str3d << "." << endl;
    str1d = str3d;
    cout << "The string str1 reassigned with string str3d is: "
        << str1d << "." << endl << endl;

    // The fifth member function assigning a specific
    // number of characters of a certain value to a string
    string str1e ( "Hello " );
    str1e.assign ( 4 , '!' );
    cout << "The string str1 assigned with exclamations is: "
        << str1e << endl << endl;

    // The sixth member function assigning the value from
    // the range of one string to another string
    string str1f ( "Hello " ), str2f ( "Wide World " );
    cout << "The string str2f is: " << str2f << endl;
    str1f.assign ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
    cout << "The string str1 assigned a range of string str2f is: "
        << str1f << "." << endl << endl;
}

```

```
The C-string cstr1a is: Out There.
Assigning the C-string cstr1a to string str1 gives: Out There.

The C-string cstr1b is: Out There
Assigning the 1st part of the C-string cstr1b to string str1 gives: Out.

The string str2c is: Wide World
The newly assigned string str1 is: World.

The original string str1 is: Hello.
The string str2d is: Wide
The string str1 newly assigned with string str2d is: Wide.
The string str3d is: World.
The string str1 reassigned with string str3d is: World.

The string str1 assigned with exclamations is: !!!!

The string str2f is: Wide World
The string str1 assigned a range of string str2f is: World.
```

basic_string::at

Provides a reference to the character with a specified index in a string.

```
const_reference at(size_type _Off) const;

reference at(size_type _Off);
```

Parameters

_Off

The index of the position of the element to be referenced.

Return Value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero and the following elements are indexed consecutively by the positive integers, so that a string of length n has an n th element indexed by the number $n - 1$.

The member [operator\[\]](#) is faster than the member function `at` for providing read and write access to the elements of a string.

The member `operator[]` does not check whether the index passed as a parameter is valid but the member function `at` does and so should be used if the validity is not certain. An invalid index, which is an index less than zero or greater than or equal to the size of the string, passed to the member function `at` throws an [out_of_range Class](#) exception. An invalid index passed to the `operator[]` results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null-character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non- **const** strings.

Example

```

// basic_string_at.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Hello world" ), str2 ( "Goodbye world" );
    const string  cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
    cout << "The original string str1 is: " << str1 << endl;
    cout << "The original string str2 is: " << str2 << endl;

    // Element access to the non const strings
    basic_string<char>::reference refStr1 = str1 [6];
    basic_string<char>::reference refStr2 = str2.at ( 3 );

    cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
    cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

    // Element access to the const strings
    basic_string<char>::const_reference crefStr1 = cstr1 [ cstr1.length ( ) ];
    basic_string<char>::const_reference crefStr2 = cstr2.at ( 8 );

    if ( crefStr1 == '\0' )
        cout << "The null character is returned as a valid reference."
            << endl;
    else
        cout << "The null character is not returned." << endl;
    cout << "The character with index 8 in the const string cstr2 is: "
        << crefStr2 << "." << endl;
}

```

basic_string::back

Returns a reference to the last element in the string.

```

const_reference back() const;

reference back();

```

Return Value

A reference to the last element of the string, which must be non-empty.

Remarks

basic_string::basic_string

Constructs a string that is empty, initialized by specific characters, or is a copy of all or part of another string object or C style (null-terminated) string.

```

basic_string();

explicit basic_string(
    const allocator_type& _Al);

basic_string(
    const basic_string& right);

basic_string(
    basic_string&& right);

basic_string(
    const basic_string& right,
    size_type _Roff,
    size_type count = npos);

basic_string(
    const basic_string& right,
    size_type _Roff,
    size_type count,
    const allocator_type& _Al);

basic_string(
    const value_type* ptr,
    size_type count);

basic_string(
    const value_type* ptr,
    size_type count,
    const allocator_type& _Al);

basic_string(
    const value_type* ptr);

basic_string(
    const value_type* ptr,
    const allocator_type& _Al);

basic_string(
    size_type count,
    value_type _Ch);

basic_string(
    size_type count,
    value_type _Ch,
    const allocator_type& _Al);

template <class InputIterator>
basic_string(
    InputIterator first,
    InputIterator last);

template <class InputIterator>
basic_string(
    InputIterator first,
    InputIterator last,
    const allocator_type& _Al);

basic_string(
    const_pointer first,
    const_pointer last);

basic_string(
    const_iterator first,
    const_iterator last);

```

Parameters

ptr

The C-string whose characters are to be used to initialize the `string` being constructed. This value cannot be a null pointer.

_Al

The storage allocator class for the string object being constructed.

count

The number of characters to be initialized.

right

The string to initialize the string being constructed.

_Roff

The index of a character in a string that is the first to be used to initialize character values for the string being constructed.

_Ch

The character value to be copied into the string being constructed.

first

An input iterator, `const_pointer`, or `const_iterator` addressing the first element in the source range to be inserted.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the position of the one beyond the last element in the source range to be inserted.

Return Value

A reference to the string object that is being constructed by the constructors.

Remarks

All constructors store an `basic_string::allocator_type` and initialize the controlled sequence. The allocator object is the argument `a1`, if present. For the copy constructor, it is `right.basic_string::get_allocator()`. Otherwise, it is `Alloc()`.

The controlled sequence is initialized to a copy of the operand sequence specified by the remaining operands. A constructor without an operand sequence specifies an empty initial controlled sequence. If `InputIterator` is an integer type in a template constructor, the operand sequence `_F first, last` behaves the same as `(size_type) first, (value_type) last`.

Example

```

// basic_string_ctor.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function initializing with a C-string
    const char *cstr1a = "Hello Out There.";
    basic_string<char> str1a ( cstr1a , 5);
    cout << "The string initialized by C-string cstr1a is: "
         << str1a << "." << endl;

    // The second member function initializing with a string
    string str2a ( "How Do You Do" );
    basic_string<char> str2b ( str2a , 7 , 7 );
    cout << "The string initialized by part of the string cstr2a is: "
         << str2b << "." << endl;

    // The third member function initializing a string
    // with a number of characters of a specific value
    basic_string<char> str3a ( 5, '9' );
    cout << "The string initialized by five number 9s is: "
         << str3a << endl;

    // The fourth member function creates an empty string
    // and string with a specified allocator
    basic_string<char> str4a;
    string str4b;
    basic_string<char> str4c ( str4b.get_allocator( ) );
    if (str4c.empty( ) )
        cout << "The string str4c is empty." << endl;
    else
        cout << "The string str4c is not empty." << endl;

    // The fifth member function initializes a string from
    // another range of characters
    string str5a ( "Hello World" );
    basic_string<char> str5b ( str5a.begin( ) + 5 , str5a.end( ) );
    cout << "The string initialized by another range is: "
         << str5b << "." << endl;
}

```

basic_string::begin

Returns an iterator addressing the first element in the string.

```

const_iterator begin() const;

iterator begin();

```

Return Value

A random-access iterator that addresses the first element of the sequence or just beyond the end of an empty sequence.

Example

```

// basic_string_begin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
    using namespace std;
    string str1 ( "No way out." ), str2;
    basic_string<char>::iterator strp_Iter, str1_Iter, str2_Iter;
    basic_string<char>::const_iterator str1_cIter;

    str1_Iter = str1.begin ( );
    cout << "The first character of the string str1 is: "
         << *str1_Iter << endl;
    cout << "The full original string str1 is: " << str1 << endl;

    // The dereferenced iterator can be used to modify a character
    *str1_Iter = 'G';
    cout << "The first character of the modified str1 is now: "
         << *str1_Iter << endl;
    cout << "The full modified string str1 is now: " << str1 << endl;

    // The following line would be an error because iterator is const
    // *str1_cIter = 'g';

    // For an empty string, begin is equivalent to end
    if ( str2.begin ( ) == str2.end ( ) )
        cout << "The string str2 is empty." << endl;
    else
        cout << "The string str2 is not empty." << endl;
}

```

basic_string::c_str

Converts the contents of a string as a C-style, null-terminated string.

```
const value_type *c_str() const;
```

Return Value

A pointer to the C-style version of the invoking string. The pointer value is not valid after calling a non-const function, including the destructor, in the `basic_string` class on the object.

Remarks

Objects of type `string` belonging to the C++ template class `basic_string<char>` are not necessarily null terminated. The null character `'\0'` is used as a special character in a C-string to mark the end of the string but has no special meaning in an object of type `string` and may be a part of the string just like any other character. There is an automatic conversion from **const char*** into strings, but the string class does not provide for automatic conversions from C-style strings to objects of type **basic_string<char>**.

The returned C-style string should not be modified, as this could invalidate the pointer to the string, or deleted, as the string has a limited lifetime and is owned by the class `string`.

Example


```

// basic_string_c_str.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    string str1 ( "Hello world" );
    cout << "The original string object str1 is: "
          << str1 << endl;
    cout << "The length of the string object str1 = "
          << str1.length ( ) << endl << endl;

    // Converting a string to an array of characters
    const char *ptr1 = 0;
    ptr1= str1.data ( );
    cout << "The modified string object ptr1 is: " << ptr1
          << endl;
    cout << "The length of character array str1 = "
          << strlen ( ptr1) << endl << endl;

    // Converting a string to a C-style string
    const char *c_str1 = str1.c_str ( );
    cout << "The C-style string c_str1 is: " << c_str1
          << endl;
    cout << "The length of C-style string str1 = "
          << strlen ( c_str1) << endl << endl;
}

```

```

The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11

The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11

```

basic_string::capacity

Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.

```
size_type capacity() const;
```

Return Value

The size of storage currently allocated in memory to hold the string.

Remarks

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as [size](#).

Example

```

// basic_string_capacity.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world");
    cout << "The original string str1 is: " << str1 << endl;

    // The size and length member functions differ in name only
    basic_string<char>::size_type sizeStr1, lenStr1;
    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );

    basic_string<char>::size_type capStr1, max_sizeStr1;
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}

```

basic_string::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin` , the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`basic_string::cend`

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

`basic_string::clear`

Erases all elements of a string.

```
void clear();
```

Remarks

The string on which the member function is called will be empty.

Example

```

// basic_string_clear.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world"), str2;
    basic_string<char>::iterator str_Iter;
    cout << "The original string str1 is: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;

    str1.clear ( );
    cout << "The modified string str1 is: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;

    //For an empty string, begin is equivalent to end
    if ( str1.begin ( ) == str1.end ( ) )
        cout << "Nothing printed above because "
            << "the string str1 is empty." << endl;
    else
        cout << "The string str1 is not empty." << endl;
}

```

```

The original string str1 is: Hello world
The modified string str1 is:
Nothing printed above because the string str1 is empty.

```

basic_string::compare

Performs a case sensitive comparison with a specified string to determine if the two strings are equal or if one is lexicographically less than the other.

```

int compare(
    const basic_string<CharType, Traits, Allocator>& str) const;

int compare(
    size_type _Pos1,
    size_type _Num1,
    const basic_string<CharType, Traits, Allocator>& str) const;

int compare(
    size_type _Pos1,
    size_type _Num1,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off,
    size_type count) const;

int compare(
    const value_type* ptr) const;

int compare(
    size_type _Pos1,
    size_type _Num1,
    const value_type* ptr) const;

int compare(
    size_type _Pos1,
    size_type _Num1,
    const value_type* ptr
    size_type _Num2) const;

```

Parameters

str

The string that is to be compared to the operand string.

_Pos1

The index of the operand string at which the comparison begins.

_Num1

The maximum number of characters from the operand string to be compared.

_Num2

The maximum number of characters from the parameter string to be compared.

_Off

The index of the parameter string at which the comparison begins.

count

The maximum number of characters from the parameter string to be compared.

ptr

The C-string to be compared to the operand string.

Return Value

A negative value if the operand string is less than the parameter string; zero if the two strings are equal; or a positive value if the operand string is greater than the parameter string.

Remarks

The `compare` member functions compare either all or part of the parameter and operand strings depending on which is used.

The comparison performed is case sensitive.

Example

```
// basic_string_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function compares
    // an operand string to a parameter string
    int comp1;
    string s1o ( "CAB" );
    string s1p ( "CAB" );
    cout << "The operand string is: " << s1o << endl;
    cout << "The parameter string is: " << s1p << endl;
    comp1 = s1o.compare ( s1p );
    if ( comp1 < 0 )
        cout << "The operand string is less than "
            << "the parameter string." << endl;
    else if ( comp1 == 0 )
        cout << "The operand string is equal to "
            << "the parameter string." << endl;
    else
        cout << "The operand string is greater than "
            << "the parameter string." << endl;
    cout << endl;

    // The second member function compares part of
    // an operand string to a parameter string
    int comp2a, comp2b;
    string s2o ( "AACAB" );
    string s2p ( "CAB" );
    cout << "The operand string is: " << s2o << endl;
    cout << "The parameter string is: " << s2p << endl;
    comp2a = s2o.compare ( 2 , 3 , s2p );
    if ( comp2a < 0 )
        cout << "The last three characters of "
            << "the operand string\n are less than "
            << "the parameter string." << endl;
    else if ( comp2a == 0 )
        cout << "The last three characters of "
            << "the operand string\n are equal to "
            << "the parameter string." << endl;
    else
        cout << "The last three characters of "
            << "the operand string\n is greater than "
            << "the parameter string." << endl;

    comp2b = s2o.compare ( 0 , 3 , s2p );
    if ( comp2b < 0 )
        cout << "The first three characters of "
            << "the operand string\n are less than "
            << "the parameter string." << endl;
    else if ( comp2b == 0 )
        cout << "The first three characters of "
            << "the operand string\n are equal to "
            << "the parameter string." << endl;
    else
        cout << "The first three characters of "
            << "the operand string\n is greater than "
            << "the parameter string." << endl;
    cout << endl;

    // The third member function compares part of
    // an operand string to part of a parameter string
```

```

int comp3a;
string s3o ( "AACAB" );
string s3p ( "DCABD" );
cout << "The operand string is: " << s3o << endl;
cout << "The parameter string is: " << s3p << endl;
comp3a = s3o.compare ( 2 , 3 , s3p , 1 , 3 );
if ( comp3a < 0 )
    cout << "The three characters from position 2 of "
        << "the operand string are less than\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
else if ( comp3a == 0 )
    cout << "The three characters from position 2 of "
        << "the operand string are equal to\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
else
    cout << "The three characters from position 2 of "
        << "the operand string is greater than\n "
        << "the 3 characters parameter string "
        << "from position 1." << endl;
cout << endl;

// The fourth member function compares
// an operand string to a parameter C-string
int comp4a;
string s4o ( "ABC" );
const char* cs4p = "DEF";
cout << "The operand string is: " << s4o << endl;
cout << "The parameter C-string is: " << cs4p << endl;
comp4a = s4o.compare ( cs4p );
if ( comp4a < 0 )
    cout << "The operand string is less than "
        << "the parameter C-string." << endl;
else if ( comp4a == 0 )
    cout << "The operand string is equal to "
        << "the parameter C-string." << endl;
else
    cout << "The operand string is greater than "
        << "the parameter C-string." << endl;
cout << endl;

// The fifth member function compares part of
// an operand string to a parameter C-string
int comp5a;
string s5o ( "AACAB" );
const char* cs5p = "CAB";
cout << "The operand string is: " << s5o << endl;
cout << "The parameter string is: " << cs5p << endl;
comp5a = s5o.compare ( 2 , 3 , s2p );
if ( comp5a < 0 )
    cout << "The last three characters of "
        << "the operand string\n are less than "
        << "the parameter C-string." << endl;
else if ( comp5a == 0 )
    cout << "The last three characters of "
        << "the operand string\n are equal to "
        << "the parameter C-string." << endl;
else
    cout << "The last three characters of "
        << "the operand string\n is greater than "
        << "the parameter C-string." << endl;
cout << endl;

// The sixth member function compares part of
// an operand string to part of an equal length of
// a parameter C-string
int comp6a;
string s6o ( "AACAB" );

```

```

const char* cs6p = "ACAB";
cout << "The operand string is: " << s6o << endl;
cout << "The parameter C-string is: " << cs6p << endl;
comp6a = s6o.compare ( 1 , 3 , cs6p , 3 );
if ( comp6a < 0 )
    cout << "The 3 characters from position 1 of "
        << "the operand string are less than\n "
        << "the first 3 characters of the parameter C-string."
        << endl;
else if ( comp6a == 0 )
    cout << "The 3 characters from position 2 of "
        << "the operand string are equal to\n "
        << "the first 3 characters of the parameter C-string."
        << endl;
else
    cout << "The 3 characters from position 2 of "
        << "the operand string is greater than\n "
        << "the first 3 characters of the parameter C-string."
        << endl;
cout << endl;
}

```

The operand string is: CAB
 The parameter string is: CAB
 The operand string is equal to the parameter string.

The operand string is: AACAB
 The parameter string is: CAB
 The last three characters of the operand string
 are equal to the parameter string.
 The first three characters of the operand string
 are less than the parameter string.

The operand string is: AACAB
 The parameter string is: DCABD
 The three characters from position 2 of the operand string are equal to
 the 3 characters parameter string from position 1.

The operand string is: ABC
 The parameter C-string is: DEF
 The operand string is less than the parameter C-string.

The operand string is: AACAB
 The parameter string is: CAB
 The last three characters of the operand string
 are equal to the parameter C-string.

The operand string is: AACAB
 The parameter C-string is: ACAB
 The 3 characters from position 2 of the operand string are equal to
 the first 3 characters of the parameter C-string.

basic_string::const_iterator

A type that provides a random-access iterator that can access and read a **const** element in the string.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for [begin](#) for an example of how to declare and use `const_iterator`.

basic_string::const_pointer

A type that provides a pointer to a **const** element in a string.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

The type is a synonym for `allocator_type::const_pointer`.

For type `string`, it is equivalent to `char*`.

Pointers that are declared const must be initialized when they are declared. Const pointers always point to the same memory location and may point to constant or nonconstant data.

Example

```
// basic_string_const_ptr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    basic_string<char>::const_pointer pstr1a = "In Here";
    const char *cstr1c = "Out There";

    cout << "The string pstr1a is: " << pstr1a << "." << endl;
    cout << "The C-string cstr1c is: " << cstr1c << "." << endl;
}
```

```
The string pstr1a is: In Here.
The C-string cstr1c is: Out There.
```

basic_string::const_reference

A type that provides a reference to a **const** element stored in a string for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

The type is a synonym for `allocator_type::const_reference`. For string `type`, it is equivalent to `const char&`.

Example

See the example for [at](#) for an example of how to declare and use `const_reference`.

basic_string::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the string.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for [rbegin](#) for an example of how to declare and use `const_reverse_iterator`.

basic_string::copy

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using [basic_string::_Copy_s](#) instead.

```
size_type copy(  
    value_type* ptr,  
    size_type count,  
    size_type _Off = 0) const;
```

Parameters

ptr

The target character array to which the elements are to be copied.

_Count The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

Example

```

// basic_string_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Hello World" );
    basic_string<char>::iterator str_Iter;
    char array1 [ 20 ] = { 0 };
    char array2 [ 10 ] = { 0 };
    basic_string<char>:: pointer array1Ptr = array1;
    basic_string<char>:: value_type *array2Ptr = array2;

    cout << "The original string str1 is: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;

    basic_string<char>:: size_type nArray1;
    // Note: string::copy is potentially unsafe, consider
    // using string::_Copy_s instead.
    nArray1 = str1.copy ( array1Ptr , 12 ); // C4996
    cout << "The number of copied characters in array1 is: "
        << nArray1 << endl;
    cout << "The copied characters array1 is: " << array1 << endl;

    basic_string<char>:: size_type nArray2;
    // Note: string::copy is potentially unsafe, consider
    // using string::_Copy_s instead.
    nArray2 = str1.copy ( array2Ptr , 5 , 6 ); // C4996
    cout << "The number of copied characters in array2 is: "
        << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
}

```

```

The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
The number of copied characters in array2 is: 5
The copied characters array2 is: World

```

basic_string::crbegin

Returns a const iterator that addresses the first element in a reversed string.

```
const_reverse_iterator crbegin() const;
```

Return Value

A reverse iterator that points just beyond the end of the string. The position designates the beginning of the reverse string.

basic_string::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed string.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse iterator that addresses the location succeeding the last element in a reversed string (the location that had preceded the first element in the unreversed string).

Remarks

basic_string::_Copy_s

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

```
size_type _Copy_s(  
    value_type* dest,  
    size_type dest_size,  
    size_type count,  
    size_type _Off = 0) const;
```

Parameters

dest

The target character array to which the elements are to be copied.

dest_size

The size of *dest*.

_Count The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

Example

```

// basic_string_Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1("Hello World");
    basic_string<char>::iterator str_Iter;
    const int array1_size = 20;
    char array1[array1_size] = { 0 };
    const int array2_size = 10;
    char array2[array2_size] = { 0 };
    basic_string<char>:: pointer array1Ptr = array1;
    basic_string<char>:: value_type *array2Ptr = array2;

    cout << "The original string str1 is: ";
    for (str_Iter = str1.begin(); str_Iter != str1.end(); str_Iter++)
        cout << *str_Iter;
    cout << endl;

    basic_string<char>::size_type nArray1;
    nArray1 = str1._Copy_s(array1Ptr, array1_size, 12);
    cout << "The number of copied characters in array1 is: "
        << nArray1 << endl;
    cout << "The copied characters array1 is: " << array1 << endl;

    basic_string<char>:: size_type nArray2;
    nArray2 = str1._Copy_s(array2Ptr, array2_size, 5, 6);
    cout << "The number of copied characters in array2 is: "
        << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
}

```

```

The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
The number of copied characters in array2 is: 5
The copied characters array2 is: World

```

basic_string::data

Converts the contents of a string into an array of characters.

```
const value_type *data() const;
```

Return Value

A pointer to the first element of the array containing the contents of the string, or, for an empty array, a non-null pointer that cannot be dereferenced.

Remarks

Objects of type string belonging to the C++ template class `basic_string <char>` are not necessarily null terminated. The return type for `data` is not a valid C-string, because no null character gets appended. The null character `'\0'` is used as a special character in a C-string to mark the end of the string, but has no special meaning in an object of type string and may be a part of the string object just like any other character.

There is an automatic conversion from **const char*** into strings, but the string class does not provide for automatic conversions from C-style strings to objects of type **basic_string <char>**.

The returned string should not be modified, because this could invalidate the pointer to the string, or deleted, because the string has a limited lifetime and is owned by the class string.

Example

```
// basic_string_data.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    string str1 ( "Hello world" );
    cout << "The original string object str1 is: "
          << str1 << endl;
    cout << "The length of the string object str1 = "
          << str1.length ( ) << endl << endl;

    // Converting a string to an array of characters
    const char *ptr1 = 0;
    ptr1= str1.data ( );
    cout << "The modified string object ptr1 is: " << ptr1
          << endl;
    cout << "The length of character array str1 = "
          << strlen ( ptr1) << endl << endl;

    // Converting a string to a C-style string
    const char *c_str1 = str1.c_str ( );
    cout << "The C-style string c_str1 is: " << c_str1
          << endl;
    cout << "The length of C-style string str1 = "
          << strlen ( c_str1) << endl << endl;
}
```

```
The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11

The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11
```

basic_string::difference_type

A type that provides the difference between two iterators that refer to elements within the same string.

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

For type `string`, it is equivalent to `ptrdiff_t`.

Example

```

// basic_string_diff_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "quintillion" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexChFi, indexChLi;

    indexChFi = str1.find_first_of ( "i" );
    indexChLi = str1.find_last_of ( "i" );
    basic_string<char>::difference_type diffi = indexChLi - indexChFi;

    cout << "The first character i is at position: "
        << indexChFi << "." << endl;
    cout << "The last character i is at position: "
        << indexChLi << "." << endl;
    cout << "The difference is: " << diffi << "." << endl;
}

```

```

The original string str1 is: quintillion
The first character i is at position: 2.
The last character i is at position: 8.
The difference is: 6.

```

basic_string::empty

Tests whether the string contains characters or not.

```
bool empty() const;
```

Return Value

true if the string object contains no characters; **false** if it has at least one character.

Remarks

The member function is equivalent to `size == 0`.

Example

```

// basic_string_empty.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main() {
    using namespace std;

    bool b1, b2;

    string str1 ("Hello world");
    cout << "The original string object str1 is: " << str1 << endl;
    b1 = str1.empty();
    if (b1)
        cout << "The string object str1 is empty." << endl;
    else
        cout << "The string object str1 is not empty." << endl;
    cout << endl;

    // An example of an empty string object
    string str2;
    b2 = str2.empty();
    if (b2)
        cout << "The string object str2 is empty." << endl;
    else
        cout << "The string object str2 is not empty." << endl;
}

```

basic_string::end

Returns an iterator that addresses the location succeeding the last element in a string.

```

const_iterator end() const;

iterator end();

```

Return Value

Returns a random-access iterator that addresses the location succeeding the last element in a string.

Remarks

`end` is often used to test whether an iterator has reached the end of its string. The value returned by `end` should not be dereferenced.

If the return value of `end` is assigned to a `const_iterator`, the string object cannot be modified. If the return value of `end` is assigned to an `iterator`, the string object can be modified.

Example


```

// basic_string_end.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "No way out." ), str2;
    basic_string<char>::iterator str_Iter, str1_Iter, str2_Iter;
    basic_string<char>::const_iterator str1_cIter;

    str1_Iter = str1.end ( );
    str1_Iter--;
    str1_Iter--;
    cout << "The last character-letter of the string str1 is: " << *str1_Iter << endl;
    cout << "The full original string str1 is: " << str1 << endl;

    // end used to test when an iterator has reached the end of its string
    cout << "The string is now: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;

    // The dereferenced iterator can be used to modify a character
    *str1_Iter = 'T';
    cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;
    cout << "The modified string str1 is now: " << str1 << endl;

    // The following line would be an error because iterator is const
    // *str1_cIter = 'T';

    // For an empty string, end is equivalent to begin
    if ( str2.begin( ) == str2.end ( ) )
        cout << "The string str2 is empty." << endl;
    else
        cout << "The stringstr2 is not empty." << endl;
}

```

```

The last character-letter of the string str1 is: t
The full original string str1 is: No way out.
The string is now: No way out.
The last character-letter of the modified str1 is now: T
The modified string str1 is now: No way ouT.
The string str2 is empty.

```

basic_string::erase

Removes an element or a range of elements in a string from a specified position.

```

iterator erase(
    iterator first,
    iterator last);

iterator erase(
    iterator _It);

basic_string<CharType, Traits, Allocator>& erase(
    size_type _Pos = 0,
    size_type count = npos);

```

Parameters

first

An iterator addressing the position of the first element in the range to be erased.

last

An iterator addressing the position one past the last element in the range to be erased.

_It

An iterator addressing the position of the element in the string to be erased.

_Pos

The index of the first character in the string to be removed.

count

The number of elements that will be removed if there are as many in the range of the string beginning with *_Pos*.

Return Value

For the first two member functions, an iterator addressing the first character after the last character removed by the member function. For the third member function, a reference to the string object from which the elements have been erased.

Remarks

The third member function returns ***this**.

Example

```

// basic_string_erase.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The 1st member function using a range demarcated
    // by iterators
    string str1 ( "Hello world" );
    basic_string<char>::iterator str1_Iter;
    cout << "The original string object str1 is: "
        << str1 << "." << endl;
    str1_Iter = str1.erase ( str1.begin ( ) + 3 , str1.end ( ) - 1 );
    cout << "The first element after those removed is: "
        << *str1_Iter << "." << endl;
    cout << "The modified string object str1 is: " << str1
        << "." << endl << endl;

    // The 2nd member function erasing a char pointed to
    // by an iterator
    string str2 ( "Hello World" );
    basic_string<char>::iterator str2_Iter;
    cout << "The original string object str2 is: " << str2
        << "." << endl;
    str2_Iter = str2.erase ( str2.begin ( ) + 5 );
    cout << "The first element after those removed is: "
        << *str2_Iter << "." << endl;
    cout << "The modified string object str2 is: " << str2
        << "." << endl << endl;

    // The 3rd member function erasing a number of chars
    // after a char
    string str3 ( "Hello computer" ), str3m;
    basic_string<char>::iterator str3_Iter;
    cout << "The original string object str3 is: "
        << str3 << "." << endl;
    str3m = str3.erase ( 6 , 8 );
    cout << "The modified string object str3m is: "
        << str3m << "." << endl;
}

```

```

The original string object str1 is: Hello world.
The first element after those removed is: d.
The modified string object str1 is: Held.

The original string object str2 is: Hello World.
The first element after those removed is: W.
The modified string object str2 is: HelloWorld.

The original string object str3 is: Hello computer.
The modified string object str3m is: Hello .

```

basic_string::find

Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.

```

size_type find(
    value_type _Ch,
    size_type _Off = 0) const;

size_type find(
    const value_type* ptr,
    size_type _Off = 0) const;

size_type find(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type find(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = 0) const;

```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

```

// basic_string_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "Hello Everyone" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;

    indexCh1a = str1.find ( "e" , 3 );
    if (indexCh1a != string::npos )
        cout << "The index of the 1st 'e' found after the 3rd"
            << " position in str1 is: " << indexCh1a << endl;
    else
        cout << "The character 'e' was not found in str1 ." << endl;

    indexCh1b = str1.find ( "x" );
    if (indexCh1b != string::npos )

```

```

        cout << "The index of the 'x' found in str1 is: "
              << indexCh1b << endl << endl;
    else
        cout << "The Character 'x' was not found in str1."
              << endl << endl;

    // The second member function searches a string
    // for a substring as specified by a C-string
    string str2 ( "Let me make this perfectly clear." );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "perfect";
    indexCh2a = str2.find ( cstr2 , 5 );
    if ( indexCh2a != string::npos )
        cout << "The index of the 1st element of 'perfect' "
              << "after\n the 5th position in str2 is: "
              << indexCh2a << endl;
    else
        cout << "The substring 'perfect' was not found in str2 ."
              << endl;

    const char *cstr2b = "imperfectly";
    indexCh2b = str2.find ( cstr2b , 0 );
    if (indexCh2b != string::npos )
        cout << "The index of the 1st element of 'imperfect' "
              << "after\n the 5th position in str3 is: "
              << indexCh2b << endl;
    else
        cout << "The substring 'imperfect' was not found in str2 ."
              << endl << endl;

    // The third member function searches a string
    // for a substring as specified by a C-string
    string str3 ( "This is a sample string for this program" );
    cout << "The original string str3 is: " << str3 << endl;
    basic_string<char>::size_type indexCh3a, indexCh3b;

    const char *cstr3a = "sample";
    indexCh3a = str3.find ( cstr3a );
    if ( indexCh3a != string::npos )
        cout << "The index of the 1st element of sample "
              << "in str3 is: " << indexCh3a << endl;
    else
        cout << "The substring 'perfect' was not found in str3 ."
              << endl;

    const char *cstr3b = "for";
    indexCh3b = str3.find ( cstr3b , indexCh3a + 1 , 2 );
    if (indexCh3b != string::npos )
        cout << "The index of the next occurrence of 'for' is in "
              << "str3 begins at: " << indexCh3b << endl << endl;
    else
        cout << "There is no next occurrence of 'for' in str3 ."
              << endl << endl;

    // The fourth member function searches a string
    // for a substring as specified by a string
    string str4 ( "clearly this perfectly unclear." );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string<char>::size_type indexCh4a, indexCh4b;

    string str4a ( "clear" );
    indexCh4a = str4.find ( str4a , 5 );
    if ( indexCh4a != string::npos )
        cout << "The index of the 1st element of 'clear' "
              << "after\n the 5th position in str4 is: "
              << indexCh4a << endl;
    else

```

```

else
    cout << "The substring 'clear' was not found in str4 ."
        << endl;

    string str4b ( "clear" );
    indexCh4b = str4.find ( str4b );
    if (indexCh4b != string::npos )
        cout << "The index of the 1st element of 'clear' "
            << "in str4 is: "
            << indexCh4b << endl;
    else
        cout << "The substring 'clear' was not found in str4 ."
            << endl << endl;
}

```

The original string str1 is: Hello Everyone
 The index of the 1st 'e' found after the 3rd position in str1 is: 8
 The Character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
 The index of the 1st element of 'perfect' after
 the 5th position in str2 is: 17
 The substring 'imperfect' was not found in str2 .

The original string str3 is: This is a sample string for this program
 The index of the 1st element of sample in str3 is: 10
 The index of the next occurrence of 'for' is in str3 begins at: 24

The original string str4 is: clearly this perfectly unclear.
 The index of the 1st element of 'clear' after
 the 5th position in str4 is: 25
 The index of the 1st element of 'clear' in str4 is: 0

basic_string::find_first_not_of

Searches through a string for the first character that is not an element of a specified string.

```

size_type find_first_not_of(
    value_type _Ch,
    size_type _Off = 0) const;

size_type find_first_not_of(
    const value_type* ptr,
    size_type _Off = 0) const;

size_type find_first_not_of(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type find_first_not_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = 0) const;

```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

```
// basic_string_find_first_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "xddd-1234-abcd" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;
    static const basic_string<char>::size_type npos = -1;

    indexCh1a = str1.find_first_not_of ( "d" , 2 );
    if ( indexCh1a != npos )
        cout << "The index of the 1st 'd' found after the 3rd"
              << " position in str1 is: " << indexCh1a << endl;
    else
        cout << "The character 'd' was not found in str1 ." << endl;

    indexCh1b = str1.find_first_not_of ( "x" );
    if (indexCh1b != npos )
        cout << "The index of the 'non x' found in str1 is: "
              << indexCh1b << endl << endl;
    else
        cout << "The character 'non x' was not found in str1."
              << endl << endl;

    // The second member function searches a string
    // for a substring as specified by a C-string
    string str2 ( "BBB-1111" );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "B1";
    indexCh2a = str2.find_first_not_of ( cstr2 , 6 );
    if ( indexCh2a != npos )
        cout << "The index of the 1st occurrence of an "
              << "element of 'B1' in str2 after\n the 6th "
              << "position is: " << indexCh2a << endl;
    else
        cout << "Elements of the substring 'B1' were not"
              << "\n found in str2 after the 6th position."
              << endl;

    const char *cstr2b = "B2";
    indexCh2b = str2.find_first_not_of ( cstr2b );
    if ( indexCh2b != npos )
        cout << "The index of the 1st element of 'B2' "
```

```

        << "after\n the 0th position in str2 is: "
        << indexCh2b << endl << endl;
else
    cout << "The substring 'B2' was not found in str2 ."
        << endl << endl << endl;

// The third member function searches a string
// for a substring as specified by a C-string
string str3 ( "444-555-GGG" );
cout << "The original string str3 is: " << str3 << endl;
basic_string<char>::size_type indexCh3a, indexCh3b;

const char *cstr3a = "45G";
indexCh3a = str3.find_first_not_of ( cstr3a );
if ( indexCh3a != npos )
    cout << "The index of the 1st occurrence of an "
        << "element in str3\n other than one of the "
        << "characters in '45G' is: " << indexCh3a
        << endl;
else
    cout << "Elements in str3 contain only characters "
        << " in the string '45G'. "
        << endl;

const char *cstr3b = "45G";
indexCh3b = str3.find_first_not_of ( cstr3b , indexCh3a + 1 , 2 );
if ( indexCh3b != npos )
    cout << "The index of the second occurrence of an "
        << "element of '45G' in str3\n after the 0th "
        << "position is: " << indexCh3b << endl << endl;
else
    cout << "Elements in str3 contain only characters "
        << " in the string '45G'. "
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "12-ab-12-ab" );
cout << "The original string str4 is: " << str4 << endl;
basic_string<char>::size_type indexCh4a, indexCh4b;

string str4a ( "ba3" );
indexCh4a = str4.find_first_not_of ( str4a , 5 );
if ( indexCh4a != npos )
    cout << "The index of the 1st non occurrence of an "
        << "element of 'ba3' in str4 after\n the 5th "
        << "position is: " << indexCh4a << endl;
else
    cout << "Elements other than those in the substring"
        << " 'ba3' were not found in the string str4."
        << endl;

string str4b ( "12" );
indexCh4b = str4.find_first_not_of ( str4b );
if ( indexCh4b != npos )
    cout << "The index of the 1st non occurrence of an "
        << "element of '12' in str4 after\n the 0th "
        << "position is: " << indexCh4b << endl;
else
    cout << "Elements other than those in the substring"
        << " '12' were not found in the string str4."
        << endl;
}

```


The original string str1 is: xddd-1234-abcd
The index of the 1st 'd' found after the 3rd position in str1 is: 4
The index of the 'non x' found in str1 is: 1

The original string str2 is: BBB-1111
Elements of the substring 'B1' were not found in str2 after the 6th position.
The index of the 1st element of 'B2' after the 0th position in str2 is: 3

The original string str3 is: 444-555-GGG
The index of the 1st occurrence of an element in str3 other than one of the characters in '45G' is: 3
The index of the second occurrence of an element of '45G' in str3 after the 0th position is: 7

The original string str4 is: 12-ab-12-ab
The index of the 1st non occurrence of an element of 'ba3' in str4 after the 5th position is: 5
The index of the 1st non occurrence of an element of '12' in str4 after the 0th position is: 2

basic_string::find_first_of

Searches through a string for the first character that matches any element of a specified string.

```
size_type find_first_of(
    value_type _Ch,
    size_type _Off = 0) const;

size_type find_first_of(
    const value_type* ptr,
    size_type _Off = 0) const;

size_type find_first_of(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type find_first_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = 0) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

```
// basic_string_find_first_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "abcd-1234-abcd-1234" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;
    static const basic_string<char>::size_type npos = -1;

    indexCh1a = str1.find_first_of ( "d" , 5 );
    if ( indexCh1a != npos )
        cout << "The index of the 1st 'd' found after the 5th"
              << " position in str1 is: " << indexCh1a << endl;
    else
        cout << "The character 'd' was not found in str1 ." << endl;

    indexCh1b = str1.find_first_of ( "x" );
    if ( indexCh1b != npos )
        cout << "The index of the 'x' found in str1 is: "
              << indexCh1b << endl << endl;
    else
        cout << "The character 'x' was not found in str1."
              << endl << endl;

    // The second member function searches a string
    // for any element of a substring as specified by a C-string
    string str2 ( "ABCD-1234-ABCD-1234" );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "B1";
    indexCh2a = str2.find_first_of ( cstr2 , 6 );
    if ( indexCh2a != npos )
        cout << "The index of the 1st occurrence of an "
              << "element of 'B1' in str2 after\n the 6th "
              << "position is: " << indexCh2a << endl;
    else
        cout << "Elements of the substring 'B1' were not "
              << "found in str2 after the 10th position."
              << endl;

    const char *cstr2b = "D2";
    indexCh2b = str2.find_first_of ( cstr2b );
    if ( indexCh2b != npos )
        cout << "The index of the 1st element of 'D2' "
              << "after\n the 0th position in str2 is: "
              << indexCh2b << endl << endl;
    else
        cout << "The substring 'D2' was not found in str2 ."
              << endl << endl << endl;

    // The third member function searches a string
    // for any element of a substring as specified by a C-string
    string str3 ( "123-abc-123-abc-456-EFG-456-EFG" );
    cout << "The original string str3 is: " << str3 << endl;
    basic_string<char>::size_type indexCh3a, indexCh3b;
```

```

const char *cstr3a = "5G";
indexCh3a = str3.find_first_of ( cstr3a );
if ( indexCh3a != npos )
    cout << "The index of the 1st occurrence of an "
        << "element of '5G' in str3 after\n the 0th "
        << "position is: " << indexCh3a << endl;
else
    cout << "Elements of the substring '5G' were not "
        << "found in str3\n after the 0th position."
        << endl;

const char *cstr3b = "5GF";
indexCh3b = str3.find_first_of ( cstr3b , indexCh3a + 1 , 2 );
if ( indexCh3b != npos )
    cout << "The index of the second occurrence of an "
        << "element of '5G' in str3\n after the 0th "
        << "position is: " << indexCh3b << endl << endl;
else
    cout << "Elements of the substring '5G' were not "
        << "found in str3\n after the first occurrence."
        << endl << endl;

// The fourth member function searches a string
// for any element of a substring as specified by a string
string str4 ( "12-ab-12-ab" );
cout << "The original string str4 is: " << str4 << endl;
basic_string<char>::size_type indexCh4a, indexCh4b;

string str4a ( "ba3" );
indexCh4a = str4.find_first_of ( str4a , 5 );
if ( indexCh4a != npos )
    cout << "The index of the 1st occurrence of an "
        << "element of 'ba3' in str4 after\n the 5th "
        << "position is: " << indexCh4a << endl;
else
    cout << "Elements of the substring 'ba3' were not "
        << "found in str4\n after the 0th position."
        << endl;

string str4b ( "a2" );
indexCh4b = str4.find_first_of ( str4b );
if ( indexCh4b != npos )
    cout << "The index of the 1st occurrence of an "
        << "element of 'a2' in str4 after\n the 0th "
        << "position is: " << indexCh4b << endl;
else
    cout << "Elements of the substring 'a2' were not "
        << "found in str4\n after the 0th position."
        << endl;
}

```

The original string str1 is: abcd-1234-abcd-1234
The index of the 1st 'd' found after the 5th position in str1 is: 13
The character 'x' was not found in str1.

The original string str2 is: ABCD-1234-ABCD-1234
The index of the 1st occurrence of an element of 'B1' in str2 after the 6th position is: 11
The index of the 1st element of 'D2' after the 0th position in str2 is: 3

The original string str3 is: 123-abc-123-abc-456-EFG-456-EFG
The index of the 1st occurrence of an element of '5G' in str3 after the 0th position is: 17
The index of the second occurrence of an element of '5G' in str3 after the 0th position is: 22

The original string str4 is: 12-ab-12-ab
The index of the 1st occurrence of an element of 'ba3' in str4 after the 5th position is: 9
The index of the 1st occurrence of an element of 'a2' in str4 after the 0th position is: 1

basic_string::find_last_not_of

Searches through a string for the last character that is not any element of a specified string.

```
size_type find_last_not_of(
    value_type _Ch,
    size_type _Off = npos) const;

size_type find_last_not_of(
    const value_type* ptr,
    size_type _Off = npos) const;

size_type find_last_not_of(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type find_last_not_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = npos) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to finish.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

```
// basic_string_find_last_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "dddd-1dd4-abdd" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;
    static const basic_string<char>::size_type npos = -1;

    indexCh1a = str1.find_last_not_of ( "d" , 7 );
    if ( indexCh1a != npos )
        cout << "The index of the last non 'd'\n found before the "
            << "7th position in str1 is: " << indexCh1a << endl;
    else
        cout << "The non 'd' character was not found ." << endl;

    indexCh1b = str1.find_last_not_of ( "d" );
    if ( indexCh1b != npos )
        cout << "The index of the non 'd' found in str1 is: "
            << indexCh1b << endl << endl;
    else
        cout << "The Character 'non x' was not found in str1."
            << endl << endl;

    // The second member function searches a string
    // for a substring as specified by a C-string
    string str2 ( "BBB-1111" );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "B1";
    indexCh2a = str2.find_last_not_of ( cstr2 , 6 );
    if ( indexCh2a != npos )
        cout << "The index of the last occurrence of a "
            << "element\n not of 'B1' in str2 before the 6th "
            << "position is: " << indexCh2a << endl;
    else
        cout << "Elements not of the substring 'B1' were not "
            << "\n found in str2 before the 6th position."
            << endl;

    const char *cstr2b = "B-1";
    indexCh2b = str2.find_last_not_of ( cstr2b );
    if ( indexCh2b != npos )
        cout << "The index of the last element not "
            << "in 'B-1'\n is: "
            << indexCh2b << endl << endl;
    else
        cout << "The elements of the substring 'B-1' were "
            << "not found in str2 ."
            << endl << endl;

    // The third member function searches a string
    // for a substring as specified by a C-string
    string str3 ( "444-555-GGG" );
    cout << "The original string str3 is: " << str3 << endl;
    basic_string<char>::size_type indexCh3a, indexCh3b;
```

```

const char *cstr3a = "45G";
indexCh3a = str3.find_last_not_of ( cstr3a );
if ( indexCh3a != npos )
    cout << "The index of the last occurrence of an "
        << "element in str3\n other than one of the "
        << "characters in '45G' is: " << indexCh3a
        << endl;
else
    cout << "Elements in str3 contain only characters "
        << " in the string '45G'. "
        << endl;

const char *cstr3b = "45G";
indexCh3b = str3.find_last_not_of ( cstr3b , 6 , indexCh3a - 1 );
if ( indexCh3b != npos )
    cout << "The index of the penultimate occurrence of an "
        << "element\n not in '45G' in str3 is: "
        << indexCh3b << endl << endl;
else
    cout << "Elements in str3 contain only characters "
        << " in the string '45G'. "
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "12-ab-12-ab" );
cout << "The original string str4 is: " << str4 << endl;
basic_string<char>::size_type indexCh4a, indexCh4b;

string str4a ( "b-a" );
indexCh4a = str4.find_last_not_of ( str4a , 5 );
if ( indexCh4a != npos )
    cout << "The index of the last occurrence of an "
        << "element not\n in 'b-a' in str4 before the 5th "
        << "position is: " << indexCh4a << endl;
else
    cout << "Elements other than those in the substring"
        << " 'b-a' were not found in the string str4."
        << endl;

string str4b ( "12" );
indexCh4b = str4.find_last_not_of ( str4b );
if ( indexCh4b != npos )
    cout << "The index of the last occurrence of an "
        << "element not in '12'\n in str4 before the end "
        << "position is: " << indexCh4b << endl;
else
    cout << "Elements other than those in the substring"
        << " '12'\n were not found in the string str4."
        << endl;
}

```

The original string str1 is: dddd-1dd4-abdd
The index of the last non 'd'
found before the 7th position in str1 is: 5
The index of the non 'd' found in str1 is: 11

The original string str2 is: BBB-1111
The index of the last occurrence of a element
not of 'B1' in str2 before the 6th position is: 3
The elements of the substring 'B-1' were not found in str2 .

The original string str3 is: 444-555-GGG
The index of the last occurrence of an element in str3
other than one of the characters in '45G' is: 7
The index of the penultimate occurrence of an element
not in '45G' in str3 is: 3

The original string str4 is: 12-ab-12-ab
The index of the last occurrence of an element not
in 'b-a' in str4 before the 5th position is: 1
The index of the last occurrence of an element not in '12'
in str4 before the end position is: 10

basic_string::find_last_of

Searches through a string for the last character that matches any element of a specified string.

```
size_type find_last_of(
    value_type _Ch,
    size_type _Off = npos) const;

size_type find_last_of(
    const value_type* ptr,
    size_type _Off = npos) const;

size_type find_last_of(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type find_last_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = npos) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to finish.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the last character of the substring searched for when successful; otherwise `npos`.

Example

```
// basic_string_find_last_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "abcd-1234-abcd-1234" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;
    static const basic_string<char>::size_type npos = -1;

    indexCh1a = str1.find_last_of ( "d" , 14 );
    if ( indexCh1a != npos )
        cout << "The index of the last 'd' found before the 14th"
              << " position in str1 is: " << indexCh1a << endl;
    else
        cout << "The character 'd' was not found in str1 ." << endl;

    indexCh1b = str1.find_first_of ( "x" );
    if ( indexCh1b != npos )
        cout << "The index of the 'x' found in str1 is: "
              << indexCh1b << endl << endl;
    else
        cout << "The character 'x' was not found in str1."
              << endl << endl;

    // The second member function searches a string
    // for a substring as specified by a C-string
    string str2 ( "ABCD-1234-ABCD-1234" );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "B1";
    indexCh2a = str2.find_last_of ( cstr2 , 12 );
    if ( indexCh2a != npos )
        cout << "The index of the last occurrence of an "
              << "element of 'B1' in str2 before\n the 12th "
              << "position is: " << indexCh2a << endl;
    else
        cout << "Elements of the substring 'B1' were not "
              << "found in str2 before the 12th position."
              << endl;

    const char *cstr2b = "D2";
    indexCh2b = str2.find_last_of ( cstr2b );
    if ( indexCh2b != npos )
        cout << "The index of the last element of 'D2' "
              << "after\n the 0th position in str2 is: "
              << indexCh2b << endl << endl;
    else
        cout << "The substring 'D2' was not found in str2 ."
              << endl << endl << endl;

    // The third member function searches a string
    // for a substring as specified by a C-string
    string str3 ( "456-EFG-456-EFG" );
    cout << "The original string str3 is: " << str3 << endl;
    basic_string<char>::size_type indexCh3a;
```



```

const char *cstr3a = "5E";
indexCh3a = str3.find_last_of ( cstr3a , 8 , 8 );
if ( indexCh3a != npos )
    cout << "The index of the last occurrence of an "
        << "element of '5E' in str3 before\n the 8th "
        << "position is: " << indexCh3a << endl << endl;
else
    cout << "Elements of the substring '5G' were not "
        << "found in str3\n before the 8th position."
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "12-ab-12-ab" );
cout << "The original string str4 is: " << str4 << endl;
basic_string<char>::size_type indexCh4a, indexCh4b;

string str4a ( "ba3" );
indexCh4a = str4.find_last_of ( str4a , 8 );
if ( indexCh4a != npos )
    cout << "The index of the last occurrence of an "
        << "element of 'ba3' in str4 before\n the 8th "
        << "position is: " << indexCh4a << endl;
else
    cout << "Elements of the substring 'ba3' were not "
        << "found in str4\n after the 0th position."
        << endl;

string str4b ( "a2" );
indexCh4b = str4.find_last_of ( str4b );
if ( indexCh4b != npos )
    cout << "The index of the last occurrence of an "
        << "element of 'a2' in str4 before\n the 0th "
        << "position is: " << indexCh4b << endl;
else
    cout << "Elements of the substring 'a2' were not "
        << "found in str4\n after the 0th position."
        << endl;
}

```

The original string str1 is: abcd-1234-abcd-1234
 The index of the last 'd' found before the 14th position in str1 is: 13
 The character 'x' was not found in str1.

The original string str2 is: ABCD-1234-ABCD-1234
 The index of the last occurrence of an element of 'B1' in str2 before
 the 12th position is: 11
 The index of the last element of 'D2' after
 the 0th position in str2 is: 16

The original string str3 is: 456-EFG-456-EFG
 The index of the last occurrence of an element of '5E' in str3 before
 the 8th position is: 4

The original string str4 is: 12-ab-12-ab
 The index of the last occurrence of an element of 'ba3' in str4 before
 the 8th position is: 4
 The index of the last occurrence of an element of 'a2' in str4 before
 the 0th position is: 9

basic_string::front

Returns a reference to the first element in a string.

```
const_reference front() const;

reference front();
```

Return Value

A reference to the first element of the string, which must be non-empty.

Remarks

basic_string::get_allocator

Returns a copy of the allocator object used to construct the string.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the string.

Remarks

The member function returns the stored allocator object.

Allocators for the string class specify how the class manages storage. The default allocators supplied with container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// basic_string_get_allocator.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    // The following lines declare objects
    // that use the default allocator.
    string s1;
    basic_string <char> s2;
    basic_string <char, char_traits< char >, allocator< char > > s3;

    // s4 will use the same allocator class as s1
    basic_string <char> s4( s1.get_allocator ( ) );

    basic_string <char>::allocator_type xchar = s1.get_allocator( );
    // You can now call functions on the allocator class xchar used by s1
}
```

basic_string::insert

Inserts an element or a number of elements or a range of elements into the string at a specified position.

```

basic_string<CharType, Traits, Allocator>& insert(
    size_type _P0,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& insert(
    size_type _P0,
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& insert(
    size_type _P0,
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& insert(
    size_type _P0,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off,
    size_type count);

basic_string<CharType, Traits, Allocator>& insert(
    size_type _P0,
    size_type count,
    value_type _Ch);

iterator insert(
    iterator _It);

iterator insert(
    iterator _It,
    value_type _Ch)l
template <class InputIterator>
void insert(
    iterator _It,
    InputIterator first,
    InputIterator last);

void insert(
    iterator _It,
    size_type count,
    value_type _Ch);

void insert(
    iterator _It,
    const_pointer first,
    const_pointer last);

void insert(
    iterator _It,
    const_iterator first,
    const_iterator last);

```

Parameters

_P0

The index of the position behind the point of insertion the new characters.

ptr

The C-string to be wholly or partly inserted into the string.

count

The number of characters to be inserted.

str

The string to be wholly or partly inserted into the target string.

_Off

The index of the part of the source string supplying the characters to be appended.

_Ch

The character value of the elements to be inserted.

_It

An iterator addressing the position behind which a character is to be inserted.

first

An input iterator, `const_pointer`, or `const_iterator` addressing the first element in the source range to be inserted.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the position of the one beyond the last element in the source range to be inserted.

Return Value

Either a reference to the string object that is being assigned new characters by the member function or, in the case of individual character insertions, an iterator addressing the position of the character inserted, or none, depending on the particular member function.

Example

```
// basic_string_insert.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function inserting a C-string
    // at a given position
    basic_string<char> str1a ( "way" );
    const char *cstr1a = "a";
    str1a.insert ( 0, cstr1a );
    cout << "The string with a C-string inserted at position 0 is: "
         << str1a << "." << endl;

    // The second member function inserting a C-string
    // at a given position for a specified number of elements
    basic_string<char> str2a ( "Good" );
    const char *cstr2a = "Bye Bye Baby";
    str2a.insert ( 4, cstr2a ,3 );
    cout << "The string with a C-string inserted at the end is: "
         << str2a << "." << endl;

    // The third member function inserting a string
    // at a given position
    basic_string<char> str3a ( "Bye" );
    string str3b ( "Good" );
    str3a.insert ( 0, str3b );
    cout << "The string with a string inserted at position 0 is: "
         << str3a << "." << endl;

    // The fourth member function inserting part of
    // a string at a given position
    basic_string<char> str4a ( "Good " );
    string str4b ( "Bye Bye Baby" );
    str4a.insert ( 5, str4b , 8 , 4 );
    cout << "The string with part of a string inserted at position 4 is: "
         << str4a << "." << endl;
```

```

// The fifth member function inserts a number of characters
// at a specified position in the string
string str5 ( "The number is: ." );
str5.insert ( 15 , 3 , '3' );
cout << "The string with characters inserted is: "
      << str5 << endl;

// The sixth member function inserts a character
// at a specified position in the string
string str6 ( "ABCDFG" );
basic_string<char>::iterator str6_Iter = ( str6.begin ( ) + 4 );
str6.insert ( str6_Iter , 'e' );
cout << "The string with a character inserted is: "
      << str6 << endl;

// The seventh member function inserts a range
// at a specified position in the string
string str7a ( "ABCDHIJ" );
string str7b ( "abcdefgh" );
basic_string<char>::iterator str7a_Iter = (str7a.begin ( ) + 4 );
str7a.insert ( str7a_Iter , str7b.begin ( ) + 4 , str7b.end ( ) -1 );
cout << "The string with a character inserted from a range is: "
      << str7a << endl;

// The eighth member function inserts a number of
// characters at a specified position in the string
string str8 ( "ABCDHIJ" );
basic_string<char>::iterator str8_Iter = ( str8.begin ( ) + 4 );
str8.insert ( str8_Iter , 3 , 'e' );
cout << "The string with a character inserted from a range is: "
      << str8 << endl;
}

```

```

The string with a C-string inserted at position 0 is: away.
The string with a C-string inserted at the end is: GoodBye.
The string with a string inserted at position 0 is: GoodBye.
The string with part of a string inserted at position 4 is: Good Baby.
The string with characters inserted is: The number is: 333.
The string with a character inserted is: ABCDeFG
The string with a character inserted from a range is: ABCDefgHIJ
The string with a character inserted from a range is: ABCDeeeHIJ

```

basic_string::iterator

A type that provides a random-access iterator that can access and read a **const** element in the string.

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for [begin](#) for an example of how to declare and use `iterator`.

basic_string::length

Returns the current number of elements in a string.

```
size_type length() const;
```

Remarks

The member function is the same as [size](#).

Example

```
// basic_string_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world");
    cout << "The original string str1 is: " << str1 << endl;

    // The size and length member functions differ in name only
    basic_string<char>::size_type sizeStr1, lenStr1;
    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );

    basic_string<char>::size_type capStr1, max_sizeStr1;
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

basic_string::max_size

Returns the maximum number of characters a string could contain.

```
size_type max_size() const;
```

Return Value

The maximum number of characters a string could contain.

Remarks

An exception of type [length_error Class](#) is thrown when an operation produces a string with a length greater than the maximum size.

Example

```
// basic_string_max_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world");
    cout << "The original string str1 is: " << str1 << endl;

    // The size and length member functions differ in name only
    basic_string<char>::size_type sizeStr1, lenStr1;
    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );

    basic_string<char>::size_type capStr1, max_sizeStr1;
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

basic_string::npos

An unsigned integral value initialized to -1 that indicates either "not found" or "all remaining characters" when a search function fails.

```
static const size_type npos = -1;
```

Remarks

When the return value is to be checked for the `npos` value, it might not work unless the return value is of type `size_type` and not either `int` or `unsigned`.

Example

See the example for [find](#) for an example of how to declare and use `npos`.

basic_string::operator+=

Appends characters to a string.

```
basic_string<CharType, Traits, Allocator>& operator+=(  
    value_type _Ch);  
  
basic_string<CharType, Traits, Allocator>& operator+=(  
    const value_type* ptr);  
  
basic_string<CharType, Traits, Allocator>& operator+=(  
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

_Ch

The character to be appended.

ptr

The characters of the C-string to be appended.

right

The characters of the string to be appended.

Return Value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the `operator+=` or the member functions [append](#) or [push_back](#). The `operator+=` appends single-argument values while the multiple argument append member function allows a specific part of a string to be specified for adding.

Example


```

// basic_string_op_app.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // appending a single character to a string
    string str1a ( "Hello" );
    cout << "The original string str1 is: " << str1a << endl;
    str1a += '!' ;
    cout << "The string str1 appended with an exclamation is: "
        << str1a << endl << endl;

    // The second member function
    // appending a C-string to a string
    string str1b ( "Hello " );
    const char *cstr1b = "Out There";
    cout << "The C-string cstr1b is: " << cstr1b << endl;
    str1b += cstr1b;
    cout << "Appending the C-string cstr1b to string str1 gives: "
        << str1b << "." << endl << endl;

    // The third member function
    // appending one string to another in two ways,
    // comparing append and operator [ ]
    string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World" );
    cout << "The string str2d is: " << str2d << endl;
    str1d.append ( str2d );
    cout << "The appended string str1d is: "
        << str1d << "." << endl;
    str1d += str3d;
    cout << "The doubly appended string str1 is: "
        << str1d << "." << endl << endl;
}

```

```

The original string str1 is: Hello
The string str1 appended with an exclamation is: Hello!

The C-string cstr1b is: Out There
Appending the C-string cstr1b to string str1 gives: Hello Out There.

The string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended string str1 is: Hello Wide World.

```

basic_string::operator=

Assigns new character values to the contents of a string.

```
basic_string<CharType, Traits, Allocator>& operator=(  
    value_type _Ch);  
  
basic_string<CharType, Traits, Allocator>& operator=(  
    const value_type* ptr);  
  
basic_string<CharType, Traits, Allocator>& operator=(  
    const basic_string<CharType, Traits, Allocator>& right);  
  
basic_string<CharType, Traits, Allocator>& operator=(  
    const basic_string<CharType, Traits, Allocator>&& right);
```

Parameters

_Ch

The character value to be assigned.

ptr

A pointer to the characters of the C-string to be assigned to the target string.

right

The source string whose characters are to be assigned to the target string.

Return Value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings may be assigned new character values. The new value may be either a string and C-string or a single character. The `operator=` may be used if the new value can be described by a single parameter, otherwise the member function [assign](#), which has multiple parameters, may be used to specify which part of the string is to be assigned to a target string.

Example

```

// basic_string_op_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function assigning a
    // character of a certain value to a string
    string str1a ( "Hello " );
    str1a = '0';
    cout << "The string str1 assigned with the zero character is: "
         << str1a << endl << endl;

    // The second member function assigning the
    // characters of a C-string to a string
    string str1b;
    const char *cstr1b = "Out There";
    cout << "The C-string cstr1b is: " << cstr1b << "." << endl;
    str1b = cstr1b;
    cout << "Assigning the C-string cstr1a to string str1 gives: "
         << str1b << "." << endl << endl;

    // The third member function assigning the characters
    // from one string to another string in two equivalent
    // ways, comparing the assign and operator =
    string str1c ( "Hello" ), str2c ( "Wide" ), str3c ( "World" );
    cout << "The original string str1 is: " << str1c << "." << endl;
    cout << "The string str2c is: " << str2c << "." << endl;
    str1c.assign ( str2c );
    cout << "The string str1 newly assigned with string str2c is: "
         << str1c << "." << endl;
    cout << "The string str3c is: " << str3c << "." << endl;
    str1c = str3c;
    cout << "The string str1 reassigned with string str3c is: "
         << str1c << "." << endl << endl;
}

```

The string str1 assigned with the zero character is: 0

The C-string cstr1b is: Out There.

Assigning the C-string cstr1a to string str1 gives: Out There.

The original string str1 is: Hello.

The string str2c is: Wide.

The string str1 newly assigned with string str2c is: Wide.

The string str3c is: World.

The string str1 reassigned with string str3c is: World.

basic_string::operator[]

Provides a reference to the character with a specified index in a string.

```

const_reference operator[](size_type _Off) const;
reference operator[](size_type _Off);

```

Parameters

_Off

The index of the position of the element to be referenced.

Return Value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero, and the following elements are indexed consecutively by the positive integers, so that a string of length n has an n th element indexed by the number $n - 1$.

`operator[]` is faster than the member function `at` for providing read and write access to the elements of a string.

`operator[]` does not check whether the index passed as a parameter is valid, but the member function `at` does and so should be used in the validity is not certain. An invalid index (an index less than zero or greater than or equal to the size of the string) passed to the member function `at` throws an [out_of_range Class](#) exception. An invalid index passed to `operator[]` results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non- **const** strings.

When compiling with `_ITERATOR_DEBUG_LEVEL` set to 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the string. For more information, see [Checked Iterators](#).

Example

```
// basic_string_op_ref.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Hello world" ), str2 ( "Goodbye world" );
    const string cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
    cout << "The original string str1 is: " << str1 << endl;
    cout << "The original string str2 is: " << str2 << endl;

    // Element access to the non-const strings
    basic_string<char>::reference refStr1 = str1 [6];
    basic_string<char>::reference refStr2 = str2.at ( 3 );

    cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
    cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

    // Element access to the const strings
    basic_string<char>::const_reference crefStr1 = cstr1 [ cstr1.length ( ) ];
    basic_string<char>::const_reference crefStr2 = cstr2.at ( 8 );

    if ( crefStr1 == '\0' )
        cout << "The null character is returned as a valid reference."
            << endl;
    else
        cout << "The null character is not returned." << endl;
    cout << "The character with index of 8 in the const string cstr2 is: "
        << crefStr2 << "." << endl;
}
```

`basic_string::pointer`

A type that provides a pointer to a character element in a string or character array.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

The type is a synonym for `allocator_type::pointer`.

For type `string`, it is equivalent to **char***.

Example

```
// basic_string_pointer.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    basic_string<char>::pointer pstr1a = "In Here";
    char *cstr1b = "Out There";
    cout << "The string pstr1a is: " << pstr1a << "." << endl;
    cout << "The C-string cstr1b is: " << cstr1b << "." << endl;
}
```

```
The string pstr1a is: In Here.
The C-string cstr1b is: Out There.
```

basic_string::pop_back

Erases the last element of the string.

```
void pop_back();
```

Remarks

This member function effectively calls `erase(size() - 1)` to erase the last element of the sequence, which must be non-empty.

basic_string::push_back

Adds an element to the end of the string.

```
void push_back(value_type _Ch);
```

Parameters

_Ch

The character to be added to the end of the string.

Remarks

The member function effectively calls `insert(end, _Ch)`.

Example

```

// basic_string_push_back.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "abc" );
    basic_string<char>::iterator str_Iter, str1_Iter;

    cout << "The original string str1 is: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;

    // str1.push_back ( 'd' );
    str1_Iter = str1.end ( );
    str1_Iter--;
    cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;

    cout << "The modified string str1 is: ";
    for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
        cout << *str_Iter;
    cout << endl;
}

```

```

The original string str1 is: abc
The last character-letter of the modified str1 is now: c
The modified string str1 is: abc

```

basic_string::rbegin

Returns an iterator to the first element in a reversed string.

```

const_reverse_iterator rbegin() const;

reverse_iterator rbegin();

```

Return Value

Returns a random-access iterator to the first element in a reversed string, addressing what would be the last element in the corresponding unreversed string.

Remarks

`rbegin` is used with a reversed string just as `begin` is used with a string.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the string object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the string object can be modified.

`rbegin` can be used to initialize an iteration through a string backwards.

Example

```

// basic_string_rbegin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Able was I ere I saw Elba" ), str2;
    basic_string<char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
    basic_string<char>::const_reverse_iterator str1_rcIter;

    str1_rIter = str1.rbegin ( );
    // str1_rIter--;
    cout << "The first character-letter of the reversed string str1 is: "
         << *str1_rIter << endl;
    cout << "The full reversed string str1 is:\n ";
    for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
        cout << *str_rIter;
    cout << endl;

    // The dereferenced iterator can be used to modify a character
    *str1_rIter = 'A';
    cout << "The first character-letter of the modified str1 is now: "
         << *str1_rIter << endl;
    cout << "The full modified reversed string str1 is now:\n ";
    for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
        cout << *str_rIter;
    cout << endl;

    // The following line would be an error because iterator is const
    // *str1_rcIter = 'A';

    // For an empty string, begin is equivalent to end
    if ( str2.rbegin( ) == str2.rend( ) )
        cout << "The string str2 is empty." << endl;
    else
        cout << "The stringstr2  is not empty." << endl;
}

```

```

The first character-letter of the reversed string str1 is: a
The full reversed string str1 is:
ablE was I ere I saw elbA
The first character-letter of the modified str1 is now: A
The full modified reversed string str1 is now:
Ab1E was I ere I saw elbA
The string str2 is empty.

```

basic_string::reference

A type that provides a reference to an element stored in a string.

```
typedef typename allocator_type::reference reference;
```

Remarks

A type `reference` can be used to modify the value of an element.

The type is a synonym for `allocator_type::reference`.

For type `string`, it is equivalent to `chr&`.

Example

See the example for [at](#) for an example of how to declare and use `reference`.

basic_string::rend

Returns an iterator that addresses the location succeeding the last element in a reversed string.

```
const_reverse_iterator rend() const;  
  
reverse_iterator rend();
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed string.

Remarks

`rend` is used with a reversed string just as [end](#) is used with a string.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the string object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, the string object can be modified.

`rend` can be used to test whether a reverse iterator has reached the end of its string.

The value returned by `rend` should not be dereferenced.

Example


```

// basic_string_rend.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Able was I ere I saw Elba"), str2;
    basic_string<char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
    basic_string<char>::const_reverse_iterator str1_rcIter;

    str1_rIter = str1.rend( );
    str1_rIter--;
    cout << "The last character-letter of the reversed string str1 is: "
         << *str1_rIter << endl;
    cout << "The full reversed string str1 is:\n ";
    for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
        cout << *str_rIter;
    cout << endl;

    // The dereferenced iterator can be used to modify a character
    *str1_rIter = 'o';
    cout << "The last character-letter of the modified str1 is now: "
         << *str1_rIter << endl;
    cout << "The full modified reversed string str1 is now:\n ";
    for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++ )
        cout << *str_rIter;
    cout << endl;

    // The following line would be an error because iterator is const
    // *str1_rcIter = 'T';

    // For an empty string, end is equivalent to begin
    if ( str2.rbegin( ) == str2.rend( ) )
        cout << "The string str2 is empty." << endl;
    else
        cout << "The stringstr2  is not empty." << endl;
}

```

```

The last character-letter of the reversed string str1 is: A
The full reversed string str1 is:
ablE was I ere I saw elbA
The last character-letter of the modified str1 is now: o
The full modified reversed string str1 is now:
ablE was I ere I saw elbo
The string str2 is empty.

```

basic_string::replace

Replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.

```

basic_string<CharType, Traits, Allocator>& replace(
    size_type _Pos1,
    size_type _Num1,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& replace(
    size_type _Pos1,
    size_type _Num1,
    const basic_string<CharType, Traits, Allocator>& str);

```

```

basic_string<CharType, Traits, Allocator>& replace(
    size_type _Pos1,
    size_type _Num1,
    const value_type* ptr,
    size_type _Num2);

basic_string<CharType, Traits, Allocator>& replace(
    size_type _Pos1,
    size_type _Num1,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Pos2,
    size_type _Num2);

basic_string<CharType, Traits, Allocator>& replace(
    size_type _Pos1,
    size_type _Num1,
    size_type count,
    value_type _Ch);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const value_type* ptr,
    size_type _Num2);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    size_type _Num2,
    value_type _Ch);

template <class InputIterator>
basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const_pointer first,
    const_pointer last);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const_iterator first,
    const_iterator last);

```

Parameters

str

The string that is to be a source of characters for the operand string.

_Pos1

The index of the operand string at which the replacement begins.

_Num1

The maximum number of characters to be replaced in the operand string.

_Pos2

The index of the parameter string at which the copying begins.

_Num2

The maximum number of characters to be used from the parameter C-string.

ptr

The C-string that is to be a source of characters for the operand string.

_Ch

The character to be copied into the operand string.

first0

An iterator addressing the first character to be removed in the operand string.

last0

An iterator addressing the last character to be removed in the operand string.

first

An iterator, `const_pointer`, or `const_iterator` addressing the first character to be copied in the parameter string.

last

An iterator, `const_pointer`, or `const_iterator` addressing the last character to be copied in the parameter string.

count

The number of times *_Ch* is copied into the operand string.

Return Value

The operand string with the replacement made.

Example

```
// basic_string_replace.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first two member functions replace
    // part of the operand string with
    // characters from a parameter string or C-string
    string result1a, result1b;
    string s1o ( "AAAAAAA" );
    string s1p ( "BBB" );
    const char* cs1p = "CCC";
    cout << "The operand string s1o is: " << s1o << endl;
    cout << "The parameter string s1p is: " << s1p << endl;
    cout << "The parameter C-string cs1p is: " << cs1p << endl;
    result1a = s1o.replace ( 1 , 3 , s1p );
    cout << "The result of s1o.replace ( 1 , 3 , s1p )\n is "
        << "the string: " << result1a << "." << endl;
    result1b = s1o.replace ( 5 , 3 , cs1p );
    cout << "The result of s1o.replace ( 5 , 3 , cs1p )\n is "
        << "the string: " << result1b << "." << endl;
    cout << endl;

    // The third & fourth member function replace
    // part of the operand string with characters
```

```

// part of the operand string with characters
// form part of a parameter string or C-string
string result2a, result2b;
string s2o ( "AAAAAAA" );
string s2p ( "BBB" );
const char* cs2p = "CCC";
cout << "The operand string s2o is: " << s2o << endl;
cout << "The parameter string s1p is: " << s2p << endl;
cout << "The parameter C-string cs2p is: " << cs2p << endl;
result2a = s2o.replace ( 1 , 3 , s2p , 1 , 2 );
cout << "The result of s2o.replace (1, 3, s2p, 1, 2)\n is "
    << "the string: " << result2a << "." << endl;
result2b = s2o.replace ( 4 , 3 , cs2p , 1 );
cout << "The result of s2o.replace (4 ,3 ,cs2p)\n is "
    << "the string: " << result2b << "." << endl;
cout << endl;

// The fifth member function replaces
// part of the operand string with characters
string result3a;
string s3o ( "AAAAAAA" );
char ch3p = 'C';
cout << "The operand string s3o is: " << s3o << endl;
cout << "The parameter character c1p is: " << ch3p << endl;
result3a = s3o.replace ( 1 , 3 , 4 , ch3p );
cout << "The result of s3o.replace(1, 3, 4, ch3p)\n is "
    << "the string: " << result3a << "." << endl;
cout << endl;

// The sixth & seventh member functions replace
// part of the operand string, delineated with iterators,
// with a parameter string or C-string
string s4o ( "AAAAAAA" );
string s4p ( "BBB" );
const char* cs4p = "CCC";
cout << "The operand string s4o is: " << s4o << endl;
cout << "The parameter string s4p is: " << s4p << endl;
cout << "The parameter C-string cs4p is: " << cs4p << endl;
basic_string<char>::iterator IterF0, IterL0;
IterF0 = s4o.begin ( );
IterL0 = s4o.begin ( ) + 3;
string result4a, result4b;
result4a = s4o.replace ( IterF0 , IterL0 , s4p );
cout << "The result of s1o.replace (IterF0, IterL0, s4p)\n is "
    << "the string: " << result4a << "." << endl;
result4b = s4o.replace ( IterF0 , IterL0 , cs4p );
cout << "The result of s4o.replace (IterF0, IterL0, cs4p)\n is "
    << "the string: " << result4b << "." << endl;
cout << endl;

// The 8th member function replaces
// part of the operand string delineated with iterators
// with a number of characters from a parameter C-string
string s5o ( "AAAAAAAF" );
const char* cs5p = "CCCB";
cout << "The operand string s5o is: " << s5o << endl;
cout << "The parameter C-string cs5p is: " << cs5p << endl;
basic_string<char>::iterator IterF1, IterL1;
IterF1 = s5o.begin ( );
IterL1 = s5o.begin ( ) + 4;
string result5a;
result5a = s5o.replace ( IterF1 , IterL1 , cs5p , 4 );
cout << "The result of s5o.replace (IterF1, IterL1, cs4p ,4)\n is "
    << "the string: " << result5a << "." << endl;
cout << endl;

// The 9th member function replaces
// part of the operand string delineated with iterators
// with specified characters
string s6o ( "AAAAAAG" );

```

```

string s6o ( "AAAAAA" );
char ch6p = 'q';
cout << "The operand string s6o is: " << s6o << endl;
cout << "The parameter character ch6p is: " << ch6p << endl;
basic_string<char>::iterator IterF2, IterL2;
IterF2 = s6o.begin ( );
IterL2 = s6o.begin ( ) + 3;
string result6a;
result6a = s6o.replace ( IterF2 , IterL2 , 4 , ch6p );
cout << "The result of s6o.replace (IterF1, IterL1, 4, ch6p)\n is "
    << "the string: " << result6a << "." << endl;
cout << endl;

// The 10th member function replaces
// part of the operand string delineated with iterators
// with part of a parameter string delineated with iterators
string s7o ( "0000000" );
string s7p ( "PPPP" );
cout << "The operand string s7o is: " << s7o << endl;
cout << "The parameter string s7p is: " << s7p << endl;
basic_string<char>::iterator IterF3, IterL3, IterF4, IterL4;
IterF3 = s7o.begin ( ) + 1;
IterL3 = s7o.begin ( ) + 3;
IterF4 = s7p.begin ( );
IterL4 = s7p.begin ( ) + 2;
string result7a;
result7a = s7o.replace ( IterF3 , IterL3 , IterF4 , IterL4 );
cout << "The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)\n is "
    << "the string: " << result7a << "." << endl;
cout << endl;
}

```

The operand string s1o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs1p is: CCC
The result of s1o.replace (1 , 3 , s1p)
is the string: ABBBAAAA.
The result of s1o.replace (5 , 3 , cs1p)
is the string: ABBBACCC.

The operand string s2o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs2p is: CCC
The result of s2o.replace (1, 3, s2p, 1, 2)
is the string: ABBAAAA.
The result of s2o.replace (4 ,3 ,cs2p)
is the string: ABBAC.

The operand string s3o is: AAAAAAAA
The parameter character c1p is: C
The result of s3o.replace(1, 3, 4, ch3p)
is the string: ACCCCAAAA.

The operand string s4o is: AAAAAAAA
The parameter string s4p is: BBB
The parameter C-string cs4p is: CCC
The result of s1o.replace (IterF0, IterL0, s4p)
is the string: ABBBAAAA.
The result of s4o.replace (IterF0, IterL0, cs4p)
is the string: CCCAAAAA.

The operand string s5o is: AAAAAAAF
The parameter C-string cs5p is: CCCBB
The result of s5o.replace (IterF1, IterL1, cs4p ,4)
is the string: CCCBAAAF.

The operand string s6o is: AAAAAAAG
The parameter character ch6p is: q
The result of s6o.replace (IterF1, IterL1, 4, ch6p)
is the string: qqqaAAAAG.

The operand string s7o is: 0000000
The parameter string s7p is: PPPP
The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)
is the string: OPP0000.

basic_string::reserve

Sets the capacity of the string to a number at least as great as a specified number.

```
void reserve(size_type count = 0);
```

Parameters

count

The number of characters for which memory is being reserved.

Remarks

Having sufficient capacity is important because reallocations is a time-consuming process and invalidates all references, pointers, and iterators that refer to characters in a string.

The concept of capacity for objects of type strings is the same as for objects of type vector. Unlike vector, the member function `reserve` may be called to shrink the capacity of an object. The request is nonbinding and may or may not happen. As the default value for the parameter is zero, a call of `reserve` is a non-binding

request to shrink the capacity of the string to fit the number of characters currently in the string. The capacity is never reduced below the current number of characters.

Calling `reserve` is the only possible way to shrink the capacity of a string. However, as noted above, this request is nonbinding and may not happen.

Example

```
// basic_string_reserve.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world");
    cout << "The original string str1 is: " << str1 << endl;

    basic_string<char>::size_type sizeStr1, sizerStr1;
    sizeStr1 = str1.size ( );
    basic_string<char>::size_type capStr1, caprStr1;
    capStr1 = str1.capacity ( );

    // Compare size & capacity of the original string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

    // Compare size & capacity of the string
    // with added capacity
    str1.reserve ( 40 );
    sizerStr1 = str1.size ( );
    caprStr1 = str1.capacity ( );

    cout << "The string str1with augmented capacity is: "
        << str1 << endl;
    cout << "The current size of string str1 is: "
        << sizerStr1 << "." << endl;
    cout << "The new capacity of string str1 is: "
        << caprStr1 << "." << endl << endl;

    // Compare size & capacity of the string
    // with downsized capacity
    str1.reserve ( );
    basic_string<char>::size_type sizedStr1;
    basic_string<char>::size_type capdStr1;
    sizedStr1 = str1.size ( );
    capdStr1 = str1.capacity ( );

    cout << "The string str1 with downsized capacity is: "
        << str1 << endl;
    cout << "The current size of string str1 is: "
        << sizedStr1 << "." << endl;
    cout << "The reduced capacity of string str1 is: "
        << capdStr1 << "." << endl << endl;
}
```

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.

The string str1with augmented capacity is: Hello world
The current size of string str1 is: 11.
The new capacity of string str1 is: 47.

The string str1 with downsized capacity is: Hello world
The current size of string str1 is: 11.
The reduced capacity of string str1 is: 47.
```

basic_string::resize

Specifies a new size for a string, appending or erasing elements as required.

```
void resize(
    size_type count,);

void resize(
    size_type count,
    _Elem _Ch);
```

Parameters

count

The new size of the string.

_Ch

The value that appended characters are initialized with if additional elements are required.

Remarks

If the resulting size exceeds the maximum number of characters, the form throws `length_error`.

Example


```

// basic_string_resize.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Hello world" );
    cout << "The original string str1 is: " << str1 << endl;

    basic_string<char>::size_type sizeStr1;
    sizeStr1 = str1.size ( );
    basic_string<char>::size_type capStr1;
    capStr1 = str1.capacity ( );

    // Compare size & capacity of the original string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

    // Use resize to increase size by 2 elements: exclamations
    str1.resize ( str1.size ( ) + 2 , '!' );
    cout << "The resized string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    capStr1 = str1.capacity ( );

    // Compare size & capacity of a string after resizing
    cout << "The current size of resized string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The capacity of resized string str1 is: "
        << capStr1 << "." << endl << endl;

    // Use resize to increase size by 20 elements:
    str1.resize ( str1.size ( ) + 20 );
    cout << "The resized string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    capStr1 = str1.capacity ( );

    // Compare size & capacity of a string after resizing
    // note capacity increases automatically as required
    cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl << endl;

    // Use resize to downsize by 28 elements:
    str1.resize ( str1.size ( ) - 28 );
    cout << "The downsized string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    capStr1 = str1.capacity ( );

    // Compare size & capacity of a string after downsizing
    cout << "The current size of downsized string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The capacity of downsized string str1 is: "
        << capStr1 << "." << endl;
}

```

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.
```

```
The resized string str1 is: Hello world!!
The current size of resized string str1 is: 13.
The capacity of resized string str1 is: 15.
```

```
The resized string str1 is: Hello world!!
The current size of modified string str1 is: 33.
The capacity of modified string str1 is: 47.
```

```
The downsized string str1 is: Hello
The current size of downsized string str1 is: 5.
The capacity of downsized string str1 is: 47.
```

basic_string::reverse_iterator

A type that provides a reference to an element stored in a string.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` can be used to modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for [rbegin](#) for an example of how to declare and use `reverse_iterator`.

basic_string::rfind

Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.

```
size_type rfind(
    value_type _Ch,
    size_type _Off = npos) const;

size_type rfind(
    const value_type* ptr,
    size_type _Off = npos) const;

size_type rfind(
    const value_type* ptr,
    size_type _Off,
    size_type count) const;

size_type rfind(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type _Off = npos) const;
```

Parameters

_Ch

The character value for which the member function is to search.

_Off

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return Value

The index of the last occurrence, when searched backwards, of the first character of the substring when successful; otherwise `npos`.

Example

```
// basic_string_rfind.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function
    // searches for a single character in a string
    string str1 ( "Hello Everyone" );
    cout << "The original string str1 is: " << str1 << endl;
    basic_string<char>::size_type indexCh1a, indexCh1b;
    static const basic_string<char>::size_type npos = -1;

    indexCh1a = str1.rfind ( "e" , 9 );
    if ( indexCh1a != npos )
        cout << "The index of the 1st 'e' found before the 9th"
              << " position in str1 is: " << indexCh1a << endl;
    else
        cout << "The character 'e' was not found in str1 ." << endl;

    indexCh1b = str1.rfind ( "x" );
    if ( indexCh1b != npos )
        cout << "The index of the 'x' found in str1 is: "
              << indexCh1b << endl << endl;
    else
        cout << "The character 'x' was not found in str1."
              << endl << endl;

    // The second member function searches a string
    // for a substring as specified by a C-string
    string str2 ( "Let me make this perfectly clear." );
    cout << "The original string str2 is: " << str2 << endl;
    basic_string<char>::size_type indexCh2a, indexCh2b;

    const char *cstr2 = "perfect";
    indexCh2a = str2.rfind ( cstr2 , 30 );
    if ( indexCh2a != npos )
        cout << "The index of the 1st element of 'perfect' "
              << "before\n the 30th position in str2 is: "
              << indexCh2a << endl;
    else
        cout << "The substring 'perfect' was not found in str2 ."
              << endl;

    const char *cstr2b = "imperfectly";
    indexCh2b = str2.rfind ( cstr2b , 30 );
    if ( indexCh2b != npos )
```

```

11 ( indexCh2b != npos )
    cout << "The index of the 1st element of 'imperfect' "
        << "before\n the 5th position in str3 is: "
        << indexCh2b << endl;
else
    cout << "The substring 'imperfect' was not found in str2 ."
        << endl << endl;

// The third member function searches a string
// for a substring as specified by a C-string
string str3 ( "It is a nice day. I am happy." );
cout << "The original string str3 is: " << str3 << endl;
basic_string<char>::size_type indexCh3a, indexCh3b;

const char *cstr3a = "nice";
indexCh3a = str3.rfind ( cstr3a );
if ( indexCh3a != npos )
    cout << "The index of the 1st element of 'nice' "
        << "in str3 is: " << indexCh3a << endl;
else
    cout << "The substring 'nice' was not found in str3 ."
        << endl;

const char *cstr3b = "am";
indexCh3b = str3.rfind ( cstr3b , indexCh3a + 25 , 2 );
if ( indexCh3b != npos )
    cout << "The index of the next occurrence of 'am' in "
        << "str3 begins at: " << indexCh3b << endl << endl;
else
    cout << "There is no next occurrence of 'am' in str3 ."
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "This perfectly unclear." );
cout << "The original string str4 is: " << str4 << endl;
basic_string<char>::size_type indexCh4a, indexCh4b;

string str4a ( "clear" );
indexCh4a = str4.rfind ( str4a , 15 );
if ( indexCh4a != npos )
    cout << "The index of the 1st element of 'clear' "
        << "before\n the 15th position in str4 is: "
        << indexCh4a << endl;
else
    cout << "The substring 'clear' was not found in str4 "
        << "before the 15th position." << endl;

string str4b ( "clear" );
indexCh4b = str4.rfind ( str4b );
if ( indexCh4b != npos )
    cout << "The index of the 1st element of 'clear' "
        << "in str4 is: "
        << indexCh4b << endl;
else
    cout << "The substring 'clear' was not found in str4 ."
        << endl << endl;
}

```

The original string str1 is: Hello Everyone
The index of the 1st 'e' found before the 9th position in str1 is: 8
The character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
The index of the 1st element of 'perfect' before
the 30th position in str2 is: 17
The substring 'imperfect' was not found in str2 .

The original string str3 is: It is a nice day. I am happy.
The index of the 1st element of 'nice' in str3 is: 8
The index of the next occurrence of 'am' in str3 begins at: 20

The original string str4 is: This perfectly unclear.
The substring 'clear' was not found in str4 before the 15th position.
The index of the 1st element of 'clear' in str4 is: 17

basic_string::shrink_to_fit

Discards the excess capacity of the string.

```
void shrink_to_fit();
```

Remarks

This member function eliminates any unneeded storage in the container.

basic_string::size

Returns the current number of elements in a string.

```
size_type size() const;
```

Return Value

The length of the string.

Example

```

// basic_string_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ("Hello world");
    cout << "The original string str1 is: " << str1 << endl;

    // The size and length member functions differ in name only
    basic_string<char>::size_type sizeStr1, lenStr1;
    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );

    basic_string<char>::size_type capStr1, max_sizeStr1;
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}

```

basic_string::size_type

An unsigned integer type that can represent the number of elements and indices in a string.

```
typedef typename allocator_type::size_type size_type;
```

Remarks

It is equivalent to `allocator_type::size_type`.

For type `string`, it is equivalent to `size_t`.

Example

```

// basic_string_size_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1 ( "Hello world" );

    basic_string<char>::size_type sizeStr1, capStr1;
    sizeStr1 = str1.size ( );
    capStr1 = str1.capacity ( );

    cout << "The current size of string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The capacity of string str1 is: " << capStr1
         << "." << endl;
}

```

```

The current size of string str1 is: 11.
The capacity of string str1 is: 15.

```

basic_string::substr

Copies a substring of at most some number of characters from a string beginning from a specified position.

```

basic_string<CharType, Traits, Allocator> substr(
    size_type _Off = 0,
    size_type count = npos) const;

```

Parameters

_Off

An index locating the element at the position from which the copy of the string is made, with a default value of 0.

count

The number of characters that are to be copied if they are present.

Return Value

A substring object that is a copy of elements of the string operand beginning at the position specified by the first argument.

Example

```
// basic_string_substr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    string str1 ("Heterological paradoxes are persistent.");
    cout << "The original string str1 is: \n " << str1
        << endl << endl;

    basic_string<char> str2 = str1.substr ( 6 , 7 );
    cout << "The substring str1 copied is: " << str2
        << endl << endl;

    basic_string<char> str3 = str1.substr ( );
    cout << "The default substring str3 is: \n " << str3
        << "\n which is the entire original string." << endl;
}
```

```
The original string str1 is:
Heterological paradoxes are persistent.

The substring str1 copied is: logical

The default substring str3 is:
Heterological paradoxes are persistent.
which is the entire original string.
```

basic_string::swap

Exchange the contents of two strings.

```
void swap(
    basic_string<CharType, Traits, Allocator>& str);
```

Parameters

str

The source string whose elements are to be exchanged with those in the destination string.

Remarks

If the strings being swapped have the same allocator object, the `swap` member function:

- Occurs in constant time.
- Throws no exceptions.
- Invalidates no references, pointers, or iterators that designate elements in the two strings.

Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example


```
// basic_string_swap.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // Declaring an objects of type basic_string<char>
    string s1 ( "Tweedledee" );
    string s2 ( "Tweedledum" );
    cout << "Before swapping string s1 and s2:" << endl;
    cout << " The basic_string s1 = " << s1 << "." << endl;
    cout << " The basic_string s2 = " << s2 << "." << endl;

    s1.swap ( s2 );
    cout << "After swapping string s1 and s2:" << endl;
    cout << " The basic_string s1 = " << s1 << "." << endl;
    cout << " The basic_string s2 = " << s2 << "." << endl;
}
```

```
Before swapping string s1 and s2:
The basic_string s1 = Tweedledee.
The basic_string s2 = Tweedledum.
After swapping string s1 and s2:
The basic_string s1 = Tweedledum.
The basic_string s2 = Tweedledee.
```

basic_string::traits_type

A type for the character traits of the elements stored in a string.

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the second template parameter `Traits`.

For type `string`, it is equivalent to **char_traits<char>**.

Example

See the example for [copy](#) for an example of how to declare and use `traits_type`.

basic_string::value_type

A type that represents the type of characters stored in a string.

```
typedef typename allocator_type::value_type value_type;
```

Remarks

It is equivalent to `traits_type::char_type` and is equivalent to **char** for objects of type `string`.

Example

```
// basic_string_value_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    basic_string<char>::value_type ch1 = 'G';

    char ch2 = 'H';

    cout << "The character ch1 is: " << ch1 << "." << endl;
    cout << "The character ch2 is: " << ch2 << "." << endl;
}
```

```
The character ch1 is: G.
The character ch2 is: H.
```

See also

[`<string>`](#)

[Thread Safety in the C++ Standard Library](#)

char_traits Struct

5/7/2019 • 23 minutes to read • [Edit Online](#)

The char_traits struct describes attributes associated with a character.

Syntax

```
template <class CharType>
struct char_traits;
```

Parameters

CharType

The element data type.

Remarks

The template struct describes various character traits for type `CharType`. The template class `basic_string` as well as several iostream template classes, including `basic_ios`, use this information to manipulate elements of type `CharType`. Such an element type must not require explicit construction or destruction. It must supply a default constructor, a copy constructor, and an assignment operator, with the expected semantics. A bitwise copy must have the same effect as an assignment. None of the member functions of struct char_traits can throw exceptions.

Typedefs

TYPE NAME	DESCRIPTION
<code>char_type</code>	A type of character.
<code>int_type</code>	An integer type that can represent a character of type <code>char_type</code> or an end-of-file (EOF) character.
<code>off_type</code>	An integer type that can represent offsets between positions in a stream.
<code>pos_type</code>	An integer type that can represent positions in a stream.
<code>state_type</code>	A type that represents the conversion state in for multibyte characters in a stream.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Assigns one character value to another.
<code>compare</code>	Compares up to a specified number of characters in two strings.
<code>copy</code>	Copies a specified number of characters from one string to another. Deprecated. Use <code>char_traits::_Copy_s</code> instead.

MEMBER FUNCTION	DESCRIPTION
_Copy_s	Copies a specified number of characters from one string to another.
eof	Returns the end-of-file (EOF) character.
eq	Tests whether two <code>char_type</code> characters are equal.
eq_int_type	Tests whether two characters represented as <code>int_type</code> s are equal.
find	Searches for the first occurrence of a specified character in a range of characters.
length	Returns the length of a string.
lt	Tests whether one character is less than another.
move	Copies a specified number of characters in a sequence to another, possible overlapping, sequence. Deprecated. Use char_traits::Move_s instead.
_Move_s	Copies a specified number of characters in a sequence to another, possible overlapping, sequence.
not_eof	Tests whether a character is the end-of-file (EOF) character.
to_char_type	Converts an <code>int_type</code> character to the corresponding <code>char_type</code> character and returns the result.
to_int_type	Converts a <code>char_type</code> character to the corresponding <code>int_type</code> character and returns the result.

Requirements

Header: <string>

Namespace: std

char_traits::assign

Assigns one character value to another or to a range of elements in a string.

```
static void assign(char_type& _CharTo,
                  const char_type& _CharFrom);

static char_type *assign(char_type* strTo,
                        size_t _Num,
                        char_type _CharFrom);
```

Parameters

`_CharFrom` The character whose value is to be assigned.

`_CharTo`

The element that is to be assigned the character value.

strTo

The string or character array whose initial elements are to be assigned character values.

_Num

The number of elements that are going to be assigned values.

Return Value

The second member function returns a pointer to the string whose first *_Num* elements have been assigned values of *_CharFrom*.

Example

```
// char_traits_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    // The first member function assigning
    // one character value to another character
    char ChTo = 't';
    const char ChFrom = 'f';
    cout << "The initial characters ( ChTo , ChFrom ) are: ( "
         << ChTo << " , " << ChFrom << " )." << endl;
    char_traits<char>::assign ( ChTo , ChFrom );
    cout << "After assigning, the characters ( ChTo , ChFrom ) are: ( "
         << ChTo << " , " << ChFrom << " )." << endl << endl;

    // The second member function assigning
    // character values to initial part of a string
    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type* result1;
    cout << "The target string s1 is: " << s1 << endl;
    result1 = char_traits<char>::assign ( s1 , 4 , 'f' );
    cout << "The result1 = assign ( s1 , 4 , 'f' ) is: "
         << result1 << endl;
}
```

The initial characters (ChTo , ChFrom) are: (t , f).
After assigning, the characters (ChTo , ChFrom) are: (f , f).

The target string s1 is: abcd-1234-abcd
The result1 = assign (s1 , 4 , 'f') is: ffff-1234-abcd

char_traits::char_type

A type of character.

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

See the example for [copy](#) for an example of how to declare and use `char_type`.

char_traits::compare

Compares up to a specified number of characters in two strings.

```
static int compare(const char_type* str1,
                  const char_type* str2,
                  size_t _Num);
```

Parameters

str1

The first of two strings to be compared to each other.

str2

The second of two strings to be compared to each other.

_Num

The number of elements in the strings to be compared.

Return Value

A negative value if the first string is less than the second string, 0 if the two strings are equal, or a positive value if the first string is greater than the second string.

Remarks

The comparison between the strings is made element by element, first testing for equality and then, if a pair of elements in the sequence tests not equal, they are tested for less than.

If two strings compare equal over a range but one is longer than the other, then the shorter of the two is less than the longer one.

Example

```

// char_traits_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main() {
    using namespace std;

    char_traits<char>::char_type* s1 = "CAB";
    char_traits<char>::char_type* s2 = "ABC";
    char_traits<char>::char_type* s3 = "ABC";
    char_traits<char>::char_type* s4 = "ABCD";

    cout << "The string s1 is: " << s1 << endl;
    cout << "The string s2 is: " << s2 << endl;
    cout << "The string s3 is: " << s3 << endl;
    cout << "The string s4 is: " << s4 << endl;

    int comp1, comp2, comp3, comp4;
    comp1 = char_traits<char>::compare ( s1 , s2 , 2 );
    comp2 = char_traits<char>::compare ( s2 , s3 , 3 );
    comp3 = char_traits<char>::compare ( s3 , s4 , 4 );
    comp4 = char_traits<char>::compare ( s4 , s3 , 4 );
    cout << "compare ( s1 , s2 , 2 ) = " << comp1 << endl;
    cout << "compare ( s2 , s3 , 3 ) = " << comp2 << endl;
    cout << "compare ( s3 , s4 , 4 ) = " << comp3 << endl;
    cout << "compare ( s4 , s3 , 4 ) = " << comp4 << endl;
}

```

char_traits::copy

Copies a specified number of characters from one string to another.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using [char_traits::_Copy_s](#) instead.

```

static char_type *copy(char_type* _To,
    const char_type* _From,
    size_t _Num);

```

Parameters

_To

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From

The element at the beginning of the source string or character array to be copied.

_Num

The number of elements to be copied.

Return Value

The first element copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example

```

// char_traits_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type s2[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string is: " << s1 << endl;
    cout << "The destination string is: " << s2 << endl;
    // Note: char_traits::copy is potentially unsafe, consider
    // using char_traits::_Copy_s instead.
    result1 = char_traits<char>::copy ( s1 , s2 , 4 ); // C4996
    cout << "The result1 = copy ( s1 , s2 , 4 ) is: "
        << result1 << endl;
}

```

```

The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = copy ( s1 , s2 , 4 ) is: ABCD-1234-abcd

```

char_traits::_Copy_s

Copies a specified number of characters from one string to another.

```

static char_type *_Copy_s(
    char_type* dest,
    size_t dest_size,
    const char_type* _From,
    size_t count);

```

Parameters

dest

The string or character array targeted to receive the copied sequence of characters.

dest_size

The size of *dest*. If `char_type` is **char**, then this size is in bytes. If `char_type` is **wchar_t**, then this size is in words.

_From

The source string or character array to be copied.

count

The number of elements to be copied.

Return Value

The string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example


```
// char_traits_Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type s2[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string is: " << s1 << endl;
    cout << "The destination string is: " << s2 << endl;
    result1 = char_traits<char>::_Copy_s(s1,
        char_traits<char>::length(s1), s2, 4);
    cout << "The result1 = _Copy_s(s1, "
        << "char_traits<char>::length(s1), s2, 4) is: "
        << result1 << endl;
}
```

```
The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = _Copy_s(s1, char_traits<char>::length(s1), s2, 4) is: ABCD-1234-abcd
```

char_traits::eof

Returns the end-of-file (EOF) character.

```
static int_type eof();
```

Return Value

The EOF character.

Remarks

A value that represents end of file (such as EOF or WEOF).

The C++ standard states that this value must not correspond to a valid `char_type` value. The Microsoft C++ compiler enforces this constraint for type **char**, but not for type **wchar_t**. The example below demonstrates this.

Example

```
// char_traits_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::int_type int1;
    int1 = char_traits<char>::to_int_type(ch1);
    cout << "char_type ch1 is '" << ch1 << "' and corresponds to int_type "
         << int1 << "." << endl << endl;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

char_type ch1 is 'x' and corresponds to int_type 120.

The eof marker for char_traits<char> is: -1

The eof marker for char_traits<wchar_t> is: 65535

char_traits::eq

Tests whether two `char_type` characters are equal.

```
static bool eq(const char_type& _Ch1, const char_type& _Ch2);
```

Parameters

_Ch1

The first of two characters to be tested for equality.

_Ch2

The second of two characters to be tested for equality.

Return Value

true if the first character is equal to the second character; otherwise **false**.

Example

```

// char_traits_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::char_type ch2 = 'y';
    char_traits<char>::char_type ch3 = 'x';

    // Testing for equality
    bool b1 = char_traits<char>::eq ( ch1 , ch2 );
    if ( b1 )
        cout << "The character ch1 is equal "
              << "to the character ch2." << endl;
    else
        cout << "The character ch1 is not equal "
              << "to the character ch2." << endl;

    // An equivalent and alternatively test procedure
    if ( ch1 == ch3 )
        cout << "The character ch1 is equal "
              << "to the character ch3." << endl;
    else
        cout << "The character ch1 is not equal "
              << "to the character ch3." << endl;
}

```

```

The character ch1 is not equal to the character ch2.
The character ch1 is equal to the character ch3.

```

char_traits::eq_int_type

Tests whether two characters represented as `int_type`s are equal or not.

```
static bool eq_int_type(const int_type& _Ch1, const int_type& _Ch2);
```

Parameters

`_Ch1`

The first of the two characters to be tested for equality as `int_type`s.

`_Ch2`

The second of the two characters to be tested for equality as `int_type`s.

Return Value

true if the first character is equal to the second character; otherwise **false**.

Example

```

// char_traits_eq_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::char_type ch2 = 'y';
    char_traits<char>::char_type ch3 = 'x';

    // Converting from char_type to int_type
    char_traits<char>::int_type int1, int2 , int3;
    int1 =char_traits<char>:: to_int_type ( ch1 );
    int2 =char_traits<char>:: to_int_type ( ch2 );
    int3 =char_traits<char>:: to_int_type ( ch3 );

    cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

    // Testing for equality of int_type representations
    bool b1 = char_traits<char>::eq_int_type ( int1 , int2 );
    if ( b1 )
        cout << "The int_type representation of character ch1\n "
            << "is equal to the int_type representation of ch2."
            << endl;
    else
        cout << "The int_type representation of character ch1\n is "
            << "not equal to the int_type representation of ch2."
            << endl;

    // An equivalent and alternatively test procedure
    if ( int1 == int3 )
        cout << "The int_type representation of character ch1\n "
            << "is equal to the int_type representation of ch3."
            << endl;
    else
        cout << "The int_type representation of character ch1\n is "
            << "not equal to the int_type representation of ch3."
            << endl;
}

```

The char_types and corresponding int_types are:
 ch1 = x corresponding to int1 = 120.
 ch2 = y corresponding to int1 = 121.
 ch3 = x corresponding to int1 = 120.

The int_type representation of character ch1
 is not equal to the int_type representation of ch2.
 The int_type representation of character ch1
 is equal to the int_type representation of ch3.

char_traits::find

Searches for the first occurrence of a specified character in a range of characters.

```
static const char_type* find(const char_type* str,
    size_t _Num,
    const char_type& _Ch);
```

Parameters

str

The first character in the string to be searched.

_Num

The number of positions, counting from the first, in the range to be searched.

_Ch

The character to be searched for in the range.

Return Value

A pointer to the first occurrence of the specified character in the range if a match is found; otherwise, a null pointer.

Example

```
// char_traits_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    const char* s1 = "f2d-1234-abcd";
    const char* result1;
    cout << "The string to be searched is: " << s1 << endl;

    // Searching for a 'd' in the first 6 positions of string s1
    result1 = char_traits<char>::find ( s1 , 6 , 'd');
    cout << "The character searched for in s1 is: "
        << *result1 << endl;
    cout << "The string beginning with the first occurrence\n "
        << "of the character 'd' is: " << result1 << endl;

    // When no match is found the NULL value is returned
    const char* result2;
    result2 = char_traits<char>::find ( s1 , 3 , 'a');
    if ( result2 == NULL )
        cout << "The result2 of the search is NULL." << endl;
    else
        cout << "The result2 of the search is: " << result1
            << endl;
}
```

```
The string to be searched is: f2d-1234-abcd
The character searched for in s1 is: d
The string beginning with the first occurrence
of the character 'd' is: d-1234-abcd
The result2 of the search is NULL.
```

char_traits::int_type

An integer type that can represent a character of type `char_type` or an end-of-file (EOF) character.

```
typedef long int_type;
```

Remarks

It must be possible to type cast a value of type `CharType` to `int_type` then back to `CharType` without altering the original value.

Example

See the example for [eq_int_type](#) for an example of how to declare and use `int_type`.

char_traits::length

Returns the length of a string.

```
static size_t length(const char_type* str);
```

Parameters

str

The C-string whose length is to be measured.

Return Value

The number of elements in the sequence being measured, not including the null terminator.

Example

```
// char_traits_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    const char* str1= "Hello";
    cout << "The C-string str1 is: " << str1 << endl;

    size_t lenStr1;
    lenStr1 = char_traits<char>::length ( str1 );
    cout << "The length of C-string str1 is: "
         << lenStr1 << "." << endl;
}
```

```
The C-string str1 is: Hello
The length of C-string str1 is: 5.
```

char_traits::lt

Tests whether one character is less than another.

```
static bool lt(const char_type& _Ch1, const char_type& _Ch2);
```

Parameters

_Ch1

The first of two characters to be tested for less than.

_Ch2

The second of two characters to be tested for less than.

Return Value

true if the first character is less than the second character; otherwise **false**.

Example

```
// char_traits_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::char_type ch2 = 'y';
    char_traits<char>::char_type ch3 = 'z';

    // Testing for less than
    bool b1 = char_traits<char>::lt ( ch1 , ch2 );
    if ( b1 )
        cout << "The character ch1 is less than "
              << "the character ch2." << endl;
    else
        cout << "The character ch1 is not less "
              << "than the character ch2." << endl;

    // An equivalent and alternatively test procedure
    if ( ch3 < ch2 )
        cout << "The character ch3 is less than "
              << "the character ch2." << endl;
    else
        cout << "The character ch3 is not less "
              << "than the character ch2." << endl;
}
```

```
The character ch1 is less than the character ch2.
The character ch3 is not less than the character ch2.
```

char_traits::move

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using [char_traits::_Move_s](#) instead.

```
static char_type *move(char_type* _To,
    const char_type* _From,
    size_t _Num);
```

Parameters

_To

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From

The element at the beginning of the source string or character array to be copied.

_Num

The number of elements to be copied from the source string.

Return Value

The first element *_To* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

```
// char_traits_move.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type sFrom1[] = "abcd-1234-abcd";
    char_traits<char>::char_type sTo1[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string sFrom1 is: " << sFrom1 << endl;
    cout << "The destination stringsTo1 is: " << sTo1 << endl;
    // Note: char_traits::move is potentially unsafe, consider
    // using char_traits::_Move_s instead.
    result1 = char_traits<char>::move ( sTo1 , sFrom1 , 4 ); // C4996
    cout << "The result1 = move ( sTo1 , sFrom1 , 4 ) is: "
        << result1 << endl << endl;

    // When source and destination overlap
    char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
    char_traits<char>::char_type* result2;
    cout << "The source/destination string sToFrom2 is: "
        << sToFrom2 << endl;
    const char* findc = char_traits<char>::find ( sToFrom2 , 4 , 'c' );
    // Note: char_traits::move is potentially unsafe, consider
    // using char_traits::_Move_s instead.
    result2 = char_traits<char>::move ( sToFrom2 , findc , 8 ); // C4996
    cout << "The result2 = move ( sToFrom2 , findc , 8 ) is: "
        << result2 << endl;
}
```

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = move ( sTo1 , sFrom1 , 4 ) is: abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = move ( sToFrom2 , findc , 8 ) is: cd-1234-4-ABCD
```

char_traits::_Move_s

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.


```
static char_type *_Move_s(
    char_type* dest,
    size_t dest_size,
    const char_type* _From,
    size_t count);
```

Parameters

dest

The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

dest_size

The size of *dest*. If `char_type` is **char**, then this is in bytes. If `char_type` is **wchar_t**, then this is in words.

_From

The element at the beginning of the source string or character array to be copied.

count

The number of elements to be copied from the source string.

Return Value

The first element *dest* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

```
// char_traits_Move_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type sFrom1[] = "abcd-1234-abcd";
    char_traits<char>::char_type sTo1[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string sFrom1 is: " << sFrom1 << endl;
    cout << "The destination stringsTo1 is: " << sTo1 << endl;
    result1 = char_traits<char>::_Move_s(sTo1,
        char_traits<char>::length(sTo1), sFrom1, 4);
    cout << "The result1 = _Move_s(sTo1, "
        << "char_traits<char>::length(sTo1), sFrom1, 4) is: "
        << result1 << endl << endl;

    // When source and destination overlap
    char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
    char_traits<char>::char_type* result2;
    cout << "The source/destination string sToFrom2 is: "
        << sToFrom2 << endl;
    const char* findc = char_traits<char>::find(sToFrom2, 4, 'c');
    result2 = char_traits<char>::_Move_s(sToFrom2,
        char_traits<char>::length(sToFrom2), findc, 8);
    cout << "The result2 = _Move_s(sToFrom2, "
        << "char_traits<char>::length(sToFrom2), findc, 8) is: "
        << result2 << endl;
}
```

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = _Move_s(sTo1, char_traits<char>::length(sTo1), sFrom1, 4) is: abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = _Move_s(sToFrom2, char_traits<char>::length(sToFrom2), findc, 8) is: cd-1234-4-ABCD
```

char_traits::not_eof

Tests whether a character is not the end-of-file (EOF) character or is the EOF.

```
static int_type not_eof(const int_type& _Ch);
```

Parameters

_Ch

The character represented as an `int_type` to be tested for whether it is the EOF character or not.

Return Value

The `int_type` representation of the character tested, if the `int_type` of the character is not equal to that of the EOF character.

If the character `int_type` value is equal to the EOF `int_type` value, then **false**.

Example

```

// char_traits_not_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::int_type int1;
    int1 = char_traits<char>:: to_int_type ( ch1 );
    cout << "The char_type ch1 is " << ch1
         << " corresponding to int_type: "
         << int1 << "." << endl;

    // EOF member function
    char_traits <char>::int_type int2 = char_traits<char>::eof ( );
    cout << "The eofReturn is: " << int2 << endl;

    // Testing for EOF or another character
    char_traits <char>::int_type eofTest1, eofTest2;
    eofTest1 = char_traits<char>::not_eof ( int1 );
    if ( !eofTest1 )
        cout << "The eofTest1 indicates ch1 is an EOF character."
             << endl;
    else
        cout << "The eofTest1 returns: " << eofTest1
             << ", which is the character: "
             << char_traits<char>::to_char_type ( eofTest1 )
             << "." << endl;

    eofTest2 = char_traits<char>::not_eof ( int2 );
    if ( !eofTest2 )
        cout << "The eofTest2 indicates int2 is an EOF character."
             << endl;
    else
        cout << "The eofTest1 returns: " << eofTest2
             << ", which is the character: "
             << char_traits<char>::to_char_type ( eofTest2 )
             << "." << endl;
}

```

```

The char_type ch1 is x corresponding to int_type: 120.
The eofReturn is: -1
The eofTest1 returns: 120, which is the character: x.
The eofTest2 indicates int2 is an EOF character.

```

char_traits::off_type

An integer type that can represent offsets between positions in a stream.

```
typedef streamoff off_type;
```

Remarks

The type is a signed integer that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for [streamoff](#), but it has essentially the same properties as that type.

char_traits::pos_type

An integer type that can represent positions in a stream.

```
typedef streampos pos_type;
```

Remarks

The type describes an object that can store all the information needed to restore an arbitrary file-position indicator within a stream. It is typically a synonym for [streampos](#), but in any case it has essentially the same properties as that type.

char_traits::state_type

A type that represents the conversion state for multibyte characters in a stream.

```
typedef implementation-defined state_type;
```

Remarks

The type describes an object that can represent a conversion state. It is typically a synonym for `mbstate_t`, but in any case it has essentially the same properties as that type.

char_traits::to_char_type

Converts an `int_type` character to the corresponding `char_type` character and returns the result.

```
static char_type to_char_type(const int_type& _Ch);
```

Parameters

`_Ch`

The `int_type` character to be represented as a `char_type`.

Return Value

The `char_type` character corresponding to the `int_type` character.

A value of `_Ch` that cannot be represented as such yields an unspecified result.

Remarks

The conversion operations [to_int_type](#) and `to_char_type` are inverse to each other, so that:

$$\text{to_int_type} (\text{to_char_type} (x)) == x$$

for any `int_type` x and

$$\text{to_char_type} (\text{to_int_type} (x)) == x$$

for any `char_type` x .

Example

```

// char_traits_to_char_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'a';
    char_traits<char>::char_type ch2 = 'b';
    char_traits<char>::char_type ch3 = 'a';

    // Converting from char_type to int_type
    char_traits<char>::int_type int1, int2 , int3;
    int1 =char_traits<char>:: to_int_type ( ch1 );
    int2 =char_traits<char>:: to_int_type ( ch2 );
    int3 =char_traits<char>:: to_int_type ( ch3 );

    cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

    // Converting from int_type back to char_type
    char_traits<char>::char_type rec_ch1;
    rec_ch1 = char_traits<char>:: to_char_type ( int1);
    char_traits<char>::char_type rec_ch2;
    rec_ch2 = char_traits<char>:: to_char_type ( int2);

    cout << "The recovered char_types and corresponding int_types are:"
        << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
        << int1 << "."
        << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
        << int2 << "." << endl << endl;

    // Testing that the conversions are inverse operations
    bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
    if ( b1 )
        cout << "The recovered char_type of ch1"
            << " is equal to the original ch1." << endl;
    else
        cout << "The recovered char_type of ch1"
            << " is not equal to the original ch1." << endl;

    // An equivalent and alternatively test procedure
    if ( rec_ch2 == ch2 )
        cout << "The recovered char_type of ch2"
            << " is equal to the original ch2." << endl;
    else
        cout << "The recovered char_type of ch2"
            << " is not equal to the original ch2." << endl;
}

```

The `char_types` and corresponding `int_types` are:

`ch1 = a` corresponding to `int1 = 97`.

`ch2 = b` corresponding to `int1 = 98`.

`ch3 = a` corresponding to `int1 = 97`.

The recovered `char_types` and corresponding `int_types` are:

recovered `ch1 = a` from `int1 = 97`.

recovered `ch2 = b` from `int2 = 98`.

The recovered `char_type` of `ch1` is equal to the original `ch1`.

The recovered `char_type` of `ch2` is equal to the original `ch2`.

`char_traits::to_int_type`

Converts a `char_type` character to the corresponding `int_type` character and returns the result.

```
static int_type to_int_type(const char_type& _Ch);
```

Parameters

_Ch

The `char_type` character to be represented as an `int_type`.

Return Value

The `int_type` character corresponding to the `char_type` character.

Remarks

The conversion operations `to_int_type` and `to_char_type` are inverse to each other, so that:

$$\text{to_int_type} \left(\text{to_char_type} (x) \right) == x$$

for any `int_type` x , and

$$\text{to_char_type} \left(\text{to_int_type} (x) \right) == x$$

for any `char_type` x .

Example

```

// char_traits_to_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    char_traits<char>::char_type ch1 = 'a';
    char_traits<char>::char_type ch2 = 'b';
    char_traits<char>::char_type ch3 = 'a';

    // Converting from char_type to int_type
    char_traits<char>::int_type int1, int2 , int3;
    int1 =char_traits<char>:: to_int_type ( ch1 );
    int2 =char_traits<char>:: to_int_type ( ch2 );
    int3 =char_traits<char>:: to_int_type ( ch3 );

    cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

    // Converting from int_type back to char_type
    char_traits<char>::char_type rec_ch1;
    rec_ch1 = char_traits<char>:: to_char_type ( int1);
    char_traits<char>::char_type rec_ch2;
    rec_ch2 = char_traits<char>:: to_char_type ( int2);

    cout << "The recovered char_types and corresponding int_types are:"
        << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
        << int1 << "."
        << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
        << int2 << "." << endl << endl;

    // Testing that the conversions are inverse operations
    bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
    if ( b1 )
        cout << "The recovered char_type of ch1"
            << " is equal to the original ch1." << endl;
    else
        cout << "The recovered char_type of ch1"
            << " is not equal to the original ch1." << endl;

    // An equivalent and alternatively test procedure
    if ( rec_ch2 == ch2 )
        cout << "The recovered char_type of ch2"
            << " is equal to the original ch2." << endl;
    else
        cout << "The recovered char_type of ch2"
            << " is not equal to the original ch2." << endl;
}

```

The `char_types` and corresponding `int_types` are:

`ch1 = a` corresponding to `int1 = 97`.

`ch2 = b` corresponding to `int1 = 98`.

`ch3 = a` corresponding to `int1 = 97`.

The recovered `char_types` and corresponding `int_types` are:

recovered `ch1 = a` from `int1 = 97`.

recovered `ch2 = b` from `int2 = 98`.

The recovered `char_type` of `ch1` is equal to the original `ch1`.

The recovered `char_type` of `ch2` is equal to the original `ch2`.

See also

[Thread Safety in the C++ Standard Library](#)

char_traits<char> Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type **char**.

Syntax

```
template <>
struct char_traits<char>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type **char**.

Example

See the typedefs and member functions of the template class [char_traits Class](#)

char_traits<char16_t> Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type `char16_t`.

Syntax

```
template <>
struct char_traits<char16_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of the type `char16_t`.

Requirements

Header: `<string>`

Namespace: `std`

See also

[<string>](#)

[char_traits Struct](#)

[Thread Safety in the C++ Standard Library](#)

char_traits<char32_t> Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type `char32_t`.

Syntax

```
template <>
struct char_traits<char32_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type `char32_t`.

Requirements

Header: <string>

Namespace: std

See also

[<string>](#)

[char_traits Struct](#)

char_traits<wchar_t> Struct

10/31/2018 • 2 minutes to read • [Edit Online](#)

A class that is a specialization of the template struct **char_traits<CharType>** to an element of type **wchar_t**.

Syntax

```
template <>
struct char_traits<wchar_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type **wchar_t**.

Requirements

Header: <string>

Namespace: std

See also

[char_traits Struct](#)

[Thread Safety in the C++ Standard Library](#)

<string_view>

4/24/2019 • 2 minutes to read • [Edit Online](#)

Defines the class template `basic_string_view` and related types and operators. (Requires compiler option `std:c++17` or later.)

Syntax

```
#include <string_view>
```

Remarks

The `string_view` family of template specializations provides an efficient way to pass a read-only, exception-safe, non-owning handle to the character data of any string-like objects with the first element of the sequence at position zero. A function parameter of type `string_view` (which is a typedef for `basic_string_view<char>`) can accept arguments such as `std::string`, `char*`, or any other string-like class of narrow characters for which an implicit conversion to `string_view` is defined. Similarly, a parameter of `wstring_view`, `u16string_view` or `u32string_view` can accept any string type for which an implicit conversion is defined. For more information, see [basic_string_view Class](#).

Typedefs

TYPE NAME	DESCRIPTION
<code>string_view</code>	A specialization of the class template <code>basic_string_view</code> with elements of type <code>char</code> .
<code>wstring_view</code>	A specialization of the class template <code>basic_string_view</code> with elements of type <code>wchar_t</code> .
<code>u16string_view</code>	A specialization of the class template <code>basic_string_view</code> with elements of type <code>char16_t</code> .
<code>u32string_view</code>	A specialization of the class template <code>basic_string_view</code> with elements of type <code>char32_t</code> .

Operators

The `<string_view>` operators can compare `string_view` objects to objects of any convertible string types.

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the object on the left side of the operator is not equal to the object on the right side.
<code>operator==</code>	Tests if the object on the left side of the operator is equal to the object on the right side.
<code>operator<</code>	Tests if the object on the left side of the operator is less than to the object on the right side.

OPERATOR	DESCRIPTION
operator<=	Tests if the object on the left side of the operator is less than or equal to the object on the right side.
operator<<	A template function that inserts a <code>string_view</code> into an output stream.
operator>	Tests if the object on the left side of the operator is greater than to the object on the right side.
operator>=	Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

Literals

OPERATOR	DESCRIPTION
sv	Constructs a <code>string_view</code> , <code>wstring_view</code> , <code>u16string_view</code> , or <code>u32string_view</code> depending on the type of the string literal to which it is appended.

Classes

CLASS	DESCRIPTION
basic_string_view Class	A class template that provides a read-only view into a sequence of arbitrary character-like objects.
hash	Function object that produces a hash value for a <code>string_view</code> .

Requirements

- **Header:** `<string_view>`
- **Namespace:** `std`
- **Compiler Option:** `std:c++17` (or later)

See also

[Header Files Reference](#)

<string_view> operators

4/24/2019 • 5 minutes to read • [Edit Online](#)

Use these operators to compare two `string_view` objects, or a `string_view` and some other string object (for example `std::string`, or `char*`) for which an implicit conversion is provided.

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<<</code>	<code>operator<=</code>
<code>operator==</code>	<code>operator""sv</code>	

`operator!=`

Tests if the object on the left side of the operator is not equal to the object on the right side.

```
template <class CharType, class Traits>
bool operator!=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator!=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator!=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is not lexicographically equal to the object on the right side; otherwise **false**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the `string_view` on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. If they have the same number of elements and the elements are all equal, the two objects are equal. Otherwise, they are unequal.

`operator==`

Tests if the object on the left side of the operator is equal to the object on the right side.

```

template <class CharType, class Traits>
bool operator==(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator==(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator==(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);

```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is lexicographically equal to the object on the right side; otherwise **false**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the *string_view* on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. If they have the same number of elements and the elements are all equal, the two objects are equal.

operator<

Tests if the object on the left side of the operator is less than the object on the right side *string_view*

```

template <class CharType, class Traits>
bool operator<(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator<(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator<(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);

```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is lexicographically less than the object on the right side; otherwise **false**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the *string_view* on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. When the first unequal pair of characters is encountered, the result of that comparison is returned. If no unequal characters are found, but one sequence is shorter, the shorter sequence is less than the longer one. In other words, "cat" is less than "cats".

Example

```
#include <string>
#include <string_view>

using namespace std;

int main()
{
    string_view sv1 { "ABA" };
    string_view sv2{ "ABAC" };
    string_view sv3{ "ABAD" };
    string_view sv4{ "ABACE" };

    bool result = sv2 > sv1; // true
    result = sv3 > sv2; // true
    result = sv3 != sv1; // true
    result = sv4 < sv3; // true because `C` < `D`
}
```

operator<=

Tests if the object on the left side of the operator is less than or equal to the object on the right side.

```
template <class CharType, class Traits>
bool operator<=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator<=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator<=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is lexicographically less than or equal to the object on the right

side; otherwise **false**.

Remarks

See [operator<](#).

operator<<

Writes a string_view into an output stream.

```
template <class CharType, class Traits>
inline basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& Ostr, const basic_string_view<CharType, Traits> Str);
```

Parameters

Ostr

an output stream being written to.

Str

The string_view to be entered into an output stream.

Return Value

an output stream being written to.

Remarks

Use this operator to insert the contents of a string_view into an output stream, for example using [std::cout](#).

operator>

Tests if the object on the left side of the operator is greater than the object on the right side.

```
template <class CharType, class Traits>
bool operator>(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator>(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator>(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is lexicographically greater than the string_view object on the right side; otherwise **false**.

Remarks

See [operator<](#).

operator>=

Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

```
template <class CharType, class Traits>
bool operator>=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator>=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator>=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right

Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

true if the object on the left side of the operator is lexicographically greater than or equal to the object on the right side; otherwise **false**.

Remarks

See [operator<](#).

operator"" sv (string_view literal)

Constructs a string_view from a string literal. Requires namespace `std::literals::string_view_literals`.

Example

```
using namespace std;
using namespace literals::string_view_literals;

string_view sv{ "Hello"sv };
wstring_view wsv{ L"Hello"sv };
u16string_view sv16{ u"Hello"sv };
u32string_view sv32{ U"Hello"sv };
```

See also

[<string_view>](#)

<string_view> typedefs

4/24/2019 • 2 minutes to read • [Edit Online](#)

string_view	u16string_view	u32string_view
wstring_view		

string_view

A type that describes a specialization of the class template [basic_string_view](#) with elements of type **char**.

```
typedef basic_string_view<char, char_traits<char>> string_view;
```

Remarks

The following are equivalent declarations:

```
string_view str("Hello");  
  
basic_string_view<char> str("Hello");
```

For a list of string constructors, see [basic_string::basic_string](#).

u16string_view

A type that describes a specialization of the class template [basic_string_view](#) with elements of type `char16_t`.

```
typedef basic_string_view<char16_t, char_traits<char16_t>> u16string_view;
```

Remarks

For a list of string constructors, see [basic_string::basic_string](#).

u32string_view

A type that describes a specialization of the class template [basic_string_view](#) with elements of type `char32_t`.

```
typedef basic_string_view<char32_t, char_traits<char32_t>> u32string_view;
```

Remarks

For a list of string constructors, see [basic_string::basic_string](#).

wstring_view

A type that describes a specialization of the class template [basic_string_view](#) with elements of type **wchar_t**.

```
typedef basic_string_view<wchar_t, char_traits<wchar_t>> wstring_view;
```

Remarks

The following are equivalent declarations:

```
wstring_view wstr(L"Hello");  
  
basic_string_view<wchar_t> wstr(L"Hello");
```

For a list of string constructors, see [basic_string::basic_string](#).

NOTE

The size of **wchar_t** is two bytes on Windows but this is not necessarily the case for all platforms. If you need a `string_view` wide character type with a width that is guaranteed to remain the same on all platforms, use [u16string_view](#) or [u32string_view](#).

See also

[<string_view>](#)

basic_string_view Class

4/24/2019 • 21 minutes to read • [Edit Online](#)

The class template `basic_string_view<charT>` was added in C++17 to serve as a safe and efficient way for a function to accept various unrelated string types without the function having to be templated on those types. The class holds a non-owning pointer to a contiguous sequence of character data, and a length that specifies the number of characters in the sequence. No assumption is made with respect to whether the sequence is null-terminated.

The standard library defines several specializations based on the type of the elements:

- `string_view`
- `wstring_view`
- `u16string_view`
- `u32string_view`

In this document, the term "string_view" refers generally to any of these typedefs.

A `string_view` describes the minimum common interface necessary to read string data. It provides const access to the underlying data; it makes no copies (except for the `copy` function). The data may or may not contain null values (`'\0'`) at any position. A `string_view` has no control over the object's lifetime. It is the caller's responsibility to ensure that the underlying string data is valid.

A function that accepts a parameter of type `string_view` can be made to work with any string-like type, without making the function into a template, or constraining the function to a particular subset of string types. The only requirement is that an implicit conversion exists from the string type to `string_view`. All the standard string types are implicitly convertible to a `string_view` that contains the same element type. In other words, a `std::string` is convertible to a `string_view` but not to a `wstring_view`.

The following example shows a non-template function `f` that takes a parameter of type `wstring_view`. It can be called with arguments of type `std::wstring`, `wchar_t*`, and `winrt::hstring`.

```
// compile with: /std:c++17
// string_view that uses elements of wchar_t
void f(wstring_view);

// pass a std::wstring:
const std::wstring& s { L"Hello" };
f(s);

// pass a C-style null-terminated string (string_view is not null-terminated):
const wchar_t* ns = L"Hello";
f(ns);

// pass a C-style character array of len characters (excluding null terminator):
const wchar_t* cs { L"Hello" };
size_t len { 5 };
f({cs,len});

// pass a WinRT string
winrt::hstring hs { L"Hello" };
f(hs);
```

Syntax

```
template <class CharType, class Traits = char_traits<CharType>>
class basic_string_view;
```

Parameters

CharType

The type of the characters that are stored in the `string_view`. The C++ Standard Library provides the following typedefs for specializations of this template.

- [string_view](#) for elements of type **char**
- [wstring_view](#), for **wchar_t**
- [u16string_view](#) for **char16_t**
- [u32string_view](#) for **char32_t**.

Traits

Defaults to [char_traits](#)<*CharType*>.

Constructors

CONSTRUCTOR	DESCRIPTION
basic_string_view	Constructs a <code>string_view</code> that is empty or else points to all or part of some other string object's data, or to a C-style character array.

Typedefs

TYPE NAME	DESCRIPTION
const_iterator	Random-access iterator that can read const elements.
const_pointer	<pre>using const_pointer = const value_type*;</pre>
const_reference	<pre>using const_reference = const value_type&;</pre>
const_reverse_iterator	<pre>using const_reverse_iterator = std::reverse_iterator<const_iterator>;</pre>
difference_type	<pre>using difference_type = ptrdiff_t;</pre>
iterator	<pre>using iterator = const_iterator;</pre>
npos	<pre>static constexpr size_type npos = size_type(-1);</pre>
pointer	<pre>using pointer = value_type*;</pre>
reference	<pre>using reference = value_type&;</pre>
reverse_iterator	<pre>using reverse_iterator = const_reverse_iterator;</pre>
size_type	<pre>using size_type = size_t;</pre>

TYPE NAME	DESCRIPTION
traits_type	<pre>using traits_type = Traits;</pre>
value_type	<pre>using value_type = CharType;</pre>

Member operators

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns a <code>string_view</code> or convertible string object to another <code>string_view</code> .
<code>operator[]</code>	Returns the element at the specified index.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>at</code>	Returns a <code>const_reference</code> to the element at a specified location.
<code>back</code>	Returns a <code>const_reference</code> to the last element.
<code>begin</code>	Returns a <code>const</code> iterator addressing the first element. (<code>string_views</code> are immutable.)
<code>cbegin</code>	Same as <code>begin</code> .
<code>cend</code>	Returns a <code>const</code> iterator that points to one past the last element.
<code>copy</code>	Copies at most a specified number of characters from an indexed position in a source <code>string_view</code> to a target character array. (Not recommended. Use <code>_Copy_s</code> instead.)
<code>_Copy_s</code>	Secure CRT copy function.
<code>compare</code>	Compares a <code>string_view</code> with a specified <code>string_view</code> to determine if they are equal or if one is lexicographically less than the other.
<code>crbegin</code>	Same as <code>rbegin</code> .
<code>crend</code>	Same as <code>rend</code> .
<code>data</code>	Returns a raw non-owning pointer to the character sequence.
<code>empty</code>	Tests whether the <code>string_view</code> contains characters.
<code>end</code>	Same as <code>cend</code> .
<code>find</code>	Searches in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.

MEMBER FUNCTION	DESCRIPTION
find_first_not_of	Searches for the first character that is not any element of a specified <code>string_view</code> or convertible string object.
find_first_of	Searches for the first character that matches any element of a specified <code>string_view</code> or convertible string object.
find_last_not_of	Searches for the last character that is not any element of a specified <code>string_view</code> or convertible string object.
find_last_of	Searches for the last character that is an element of a specified <code>string_view</code> or convertible string object.
front	Returns a <code>const_reference</code> to the first element.
length	Returns the current number of elements.
max_size	Returns the maximum number of characters a <code>string_view</code> could contain.
rbegin	Returns a <code>const</code> iterator that addresses the first element in a reversed <code>string_view</code> .
remove_prefix	Moves the pointer forward by the specified number of elements.
remove_suffix	Reduces the size of the view by the specified number of elements starting from the back.
rend	Returns a <code>const</code> iterator that points to one past the last element in a reversed <code>string_view</code> .
rfind	Searches a <code>string_view</code> in reverse for the first occurrence of a substring that matches a specified sequence of characters.
size	Returns the current number of elements.
substr	Returns a substring of a specified length starting at a specified index.
swap	Exchange the contents of two <code>string_views</code> .

Remarks

If a function is asked to generate a sequence longer than [max_size](#) elements, the function reports a length error by throwing an object of type [length_error](#).

Requirements

[std:c++17](#) or later

Header: `<string_view>`

Namespace: `std`

basic_string_view::at

Returns a const_reference to the character at the specified 0-based index.

```
constexpr const_reference at(size_type offset) const;
```

Parameters

offset

The index of the element to be referenced.

Return Value

A const_reference to the character at the position specified by the parameter index.

Remarks

The first element has an index of zero and the following elements are indexed consecutively by the positive integers, so that a string_view of length n has an n th element indexed by the number $n - 1$. **at** throws an exception for invalid indices, unlike [operator\[\]](#).

In general, we recommend that **at** for sequences such as `std::vector` and string_view should never be used. An invalid index passed to a sequence is a logic error that should be discovered and fixed during development. If a program isn't absolutely certain that its indices are valid, it should test them, not call `at()` and rely on exceptions to defend against careless programming.

See [basic_string_view::operator\[\]](#) for more information.

Example

```
// basic_string_view_at.cpp
// compile with: /EHsc
#include <string_view>
#include <iostream>

int main()
{
    using namespace std;

    const string_view str1("Hello world");
    string_view::const_reference refStr2 = str1.at(8); // 'r'
}
```

basic_string_view::back

Returns a const_reference to the last element.

```
constexpr const_reference back() const;
```

Return Value

A const_reference to the last element in the string_view.

Remarks

Throws an exception if the string_view is empty.

Keep in mind that after a string_view is modified, for example by calling `remove_suffix`, then the element returned by this function is no longer the last element in the underlying data.

Example

A `string_view` that is constructed with a C string literal does not include the terminating null and therefore in the following example `back` returns 'p' and not '\0'.

```
char c[] = "Help"; // char[5]
string_view sv{ c };
cout << sv.size(); // size() == 4
cout << sv.back() << endl; // p
```

Embedded nulls are treated as any other character:

```
string_view e = "embedded\0nulls"sv;
cout << boolalpha << (e.back() == 's'); // true
```

basic_string_view::basic_string_view

Constructs a `string_view`.

```
constexpr basic_string_view() noexcept;
constexpr basic_string_view(const basic_string_view&) noexcept = default;
constexpr basic_string_view(const charT* str);
constexpr basic_string_view(const charT* str, size_type len);
```

Parameters

str

The pointer to the character values.

len

The number of characters to include in the view.

Remarks

The constructors with a `charT*` parameter assume that the input is null-terminated, but the terminating null is not included in the `string_view`.

You can also construct a `string_view` with a literal. See [operator""sv](#).

basic_string_view::begin

Same as [cbegin](#).

```
constexpr const_iterator begin() const noexcept;
```

Return Value

Returns a `const_iterator` addressing the first element.

basic_string_view::cbegin

Returns a `const_iterator` that addresses the first element in the range.

```
constexpr const_iterator cbegin() const noexcept;
```

Return Value

A **const** random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

basic_string_view::cend

Returns a `const_iterator` that addresses the location just beyond the last element in a range.

```
constexpr const_iterator cend() const noexcept;
```

Return Value

A **const** random-access iterator that points just beyond the end of the range.

Remarks

The value returned by `cend` should not be dereferenced.

basic_string_view::compare

Performs a case sensitive comparison with a specified `string_view` (or a convertible string type) to determine if the two objects are equal or if one is lexicographically less than the other. The [<string_view> operators](#) use this member function to perform comparisons.

```
constexpr int compare(basic_string_view strv) const noexcept;  
constexpr int compare(size_type pos, size_type num, basic_string_view strv) const;  
constexpr int compare(size_type pos, size_type num, basic_string_view strv, size_type offset, size_type num2) const;  
constexpr int compare(const charT* ptr) const;  
constexpr int compare(size_type pos, size_type num, const charT* ptr) const;  
constexpr int compare(size_type pos, size_type num, const charT* ptr, size_type num2) const;
```

Parameters

strv

The `string_view` that is to be compared to this `string_view`.

pos

The index of this `string_view` at which the comparison begins.

num

The maximum number of characters from this `string_view` to be compared.

num2

The maximum number of characters from *strv* to be compared.

offset

The index of *strv* at which the comparison begins.

ptr

The C string to be compared to this `string_view`.

Return Value

A negative value if this `string_view` is less than *strv* or *ptr*; zero if the two character sequences are equal; or a positive value if this `string_view` is greater than *strv* or *ptr*.

Remarks

The `compare` member functions perform a case-sensitive comparison of either all or part of each character sequence.

Example

```
// basic_string_view_compare.cpp
// compile with: /EHsc
#include <string_view>
#include <iostream>
#include <string>

using namespace std;

string to_alpha(int result)
{
    if (result < 0) return " less than ";
    else if (result == 0) return " equal to ";
    else return " greater than ";
}

int main()
{
    // The first member function compares
    // two string_views
    string_view sv_A("CAB");
    string_view sv_B("CAB");
    cout << "sv_A is " << sv_A << endl;
    cout << "sv_B is " << sv_B << endl;
    int comp1 = sv_A.compare(sv_B);
    cout << "sv_A is" << to_alpha(comp1) << "sv_B.\n";

    // The second member function compares part of
    // an operand string_view to another string_view
    string_view sv_C("AACAB");
    string_view sv_D("CAB");
    cout << "sv_C is: " << sv_C << endl;
    cout << "sv_D is: " << sv_D << endl;
    int comp2a = sv_C.compare(2, 3, sv_D);
    cout << "The last three characters of sv_C are"
        << to_alpha(comp2a) << "sv_D.\n";

    int comp2b = sv_C.compare(0, 3, sv_D);
    cout << "The first three characters of sv_C are"
        << to_alpha(comp2b) << "sv_D.\n";

    // The third member function compares part of
    // an operand string_view to part of another string_view
    string_view sv_E("AACAB");
    string_view sv_F("DCABD");
    cout << "sv_E: " << sv_E << endl;
    cout << "sv_F is: " << sv_F << endl;
    int comp3a = sv_E.compare(2, 3, sv_F, 1, 3);
    cout << "The three characters from position 2 of sv_E are"
        << to_alpha(comp3a)
        << "the 3 characters of sv_F from position 1.\n";

    // The fourth member function compares
    // an operand string_view to a C string
    string_view sv_G("ABC");
    const char* cs_A = "DEF";
    cout << "sv_G is: " << sv_G << endl;
    cout << "cs_A is: " << cs_A << endl;
    int comp4a = sv_G.compare(cs_A);
    cout << "sv_G is" << to_alpha(comp4a) << "cs_A.\n";

    // The fifth member function compares part of
    // an operand string_view to a C string
    string_view sv_H("AACAB");
    const char* cs_B = "CAB";
    cout << "sv_H is: " << sv_H << endl;
    cout << "cs_B is: " << cs_B << endl;
```

```

int comp5a = sv_H.compare(2, 3, cs_B);
cout << "The last three characters of sv_H are"
    << to_alpha(comp5a) << "cs_B.\n";

// The sixth member function compares part of
// an operand string_view to part of an equal length of
// a C string
string_view sv_I("AACAB");
const char* cs_C = "ACAB";
cout << "sv_I is: " << sv_I << endl;
cout << "cs_C: " << cs_C << endl;
int comp6a = sv_I.compare(1, 3, cs_C, 3);
cout << "The 3 characters from position 1 of sv_I are"
    << to_alpha(comp6a) << "the first 3 characters of cs_C.\n";
}

```

```

sv_A is CAB
sv_B is CAB
sv_A is equal to sv_B.
sv_C is: AACAB
sv_D is: CAB
The last three characters of sv_C are equal to sv_D.
The first three characters of sv_C are less than sv_D.
sv_E: AACAB
sv_F is: DCABD
The three characters from position 2 of sv_E are equal to the 3 characters of sv_F from position 1.
sv_G is: ABC
cs_A is: DEF
sv_G is less than cs_A.
sv_H is: AACAB
cs_B is: CAB
The last three characters of sv_H are equal to cs_B.
sv_I is: AACAB
cs_C: ACAB
The 3 characters from position 1 of sv_I are equal to the first 3 characters of cs_C.

```

basic_string_view::copy

Copies at most a specified number of characters from an indexed position in a source string_view to a target character array. We recommend that you use the secure function [basic_string_view::Copy_s](#) instead.

```

size_type copy(charT* ptr, size_type count, size_type offset = 0) const;

```

Parameters

ptr

The target character array to which the elements are to be copied.

count

The number of characters to be copied, at most, from the source string_view.

offset

The beginning position in the source string_view from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

basic_string_view::_Copy_s

Secure CRT copy function to be used instead of [copy](#).

```
size_type _Copy_s(  
    value_type* dest,  
    size_type dest_size,  
    size_type count,  
    size_type _Off = 0) const;
```

Parameters

dest

The target character array to which the elements are to be copied.

dest_size

The size of *dest*.

_Count The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return Value

The number of characters actually copied.

Remarks

A null character is not appended to the end of the copy.

For more information, see [c-runtime-library/security-features-in-the-crt](#).

basic_string_view::crbegin

Returns a `const_reverse_iterator` that addresses the first element in a reversed `string_view`.

```
constexpr const_reverse_iterator crbegin() const noexcept;
```

Return Value

A `const_reverse_iterator` that addresses the first element in a reversed `string_view`.

basic_string_view::crend

Same as [rend](#).

```
constexpr const_reverse_iterator crend() const noexcept;
```

Return Value

Returns a `const_reverse_iterator` that addresses one past the end of a reversed `string_view`.

basic_string_view::data

Returns a raw non-owning pointer to the `const` character sequence of the object that was used to construct the `string_view`.

```
constexpr value_type *data() const noexcept;
```

Return Value

A pointer-to-const to the first element of the character sequence.

Remarks

The pointer cannot modify the characters.

A sequence of `string_view` characters is not necessarily null-terminated. The return type for `data` is not a valid C string, because no null character gets appended. The null character `'\0'` has no special meaning in an object of type `string_view` and may be a part of the `string_view` object just like any other character.

`basic_string_view::empty`

Tests whether the `string_view` contains characters or not.

```
constexpr bool empty() const noexcept;
```

Return Value

true if the `string_view` object contains no characters; **false** if it has at least one character.

Remarks

The member function is equivalent to `size() == 0`.

`basic_string_view::end`

Returns a random-access `const_iterator` that points to one past the last element.

```
constexpr const_iterator end() const noexcept;
```

Return Value

Returns a random-access `const_iterator` that points to one past the last element.

Remarks

`end` is used to test whether a `const_iterator` has reached the end of its `string_view`. The value returned by `end` should not be dereferenced.

`basic_string_view::find`

Searches a `string_view` in a forward direction for the first occurrence of a character or substring that matches a specified sequence of character(s).

```
constexpr size_type find(basic_string_view str, size_type offset = 0) const noexcept;  
constexpr size_type find(charT chVal, size_type offset = 0) const noexcept;  
constexpr size_type find(const charT* ptr, size_type offset, size_type count) const;  
constexpr size_type find(const charT* ptr, size_type offset = 0) const;
```

Parameters

str

The `string_view` for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters in *ptr*, counting forward from the first character.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

basic_string_view::find_first_not_of

Searches for the first character that is not an element of a specified string_view or convertible string object.

```
constexpr size_type find_first_not_of(basic_string_view str, size_type offset = 0) const noexcept;
constexpr size_type find_first_not_of(charT chVal, size_type offset = 0) const noexcept;
constexpr size_type find_first_not_of(const charT* ptr, size_type offset, size_type count) const;
constexpr size_type find_first_not_of(const charT* ptr, size_type offset = 0) const;
```

Parameters

str

The string_view for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

basic_string_view::find_first_of

Searches for the first character that matches any element of a specified string_view.

```
constexpr size_type find_first_of(basic_string_view str, size_type offset = 0) const noexcept;
constexpr size_type find_first_of(charT chVal, size_type offset = 0) const noexcept;
constexpr size_type find_first_of(const charT* str, size_type offset, size_type count) const;
constexpr size_type find_first_of(const charT* str, size_type offset = 0) const;
```

Parameters

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

str

The string_view for which the member function is to search.

Return Value

The index of the first character of the substring searched for when successful; otherwise `npos`.

basic_string_view::find_last_not_of

Searches for the last character that is not any element of a specified string_view.

```
constexpr size_type find_last_not_of(basic_string_view str, size_type offset = npos) const noexcept;  
constexpr size_type find_last_not_of(charT chVal, size_type offset = npos) const noexcept;  
constexpr size_type find_last_not_of(const charT* ptr, size_type offset, size_type count) const;  
constexpr size_type find_last_not_of(const charT* ptr, size_type offset = npos) const;
```

Parameters

str

The string_view for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to finish.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in *ptr*.

Return Value

The index of the first character of the substring searched for when successful; otherwise `string_view::npos`.

basic_string_view::find_last_of

Searches for the last character that matches any element of a specified string_view.

```
constexpr size_type find_last_of(basic_string_view str, size_type offset = npos) const noexcept;  
constexpr size_type find_last_of(charT chVal, size_type offset = npos) const noexcept;  
constexpr size_type find_last_of(const charT* ptr, size_type offset, size_type count) const;  
constexpr size_type find_last_of(const charT* ptr, size_type offset = npos) const;
```

Parameters

str

The string_view for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to finish.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

Return Value

The index of the last character of the substring searched for when successful; otherwise `npos`.

basic_string_view::front

Returns a `const_reference` to the first element.

```
constexpr const_reference front() const;
```

Return Value

A `const_reference` to the first element.

Remarks

Throws an exception if the `string_view` is empty.

basic_string_view::length

Returns the current number of elements.

```
constexpr size_type length() const noexcept;
```

Remarks

The member function is the same as [size](#).

basic_string_view::max_size

Returns the maximum number of characters a `string_view` can contain.

```
constexpr size_type max_size() const noexcept;
```

Return Value

The maximum number of characters a `string_view` can contain.

Remarks

A exception of type [length_error](#) is thrown when an operation produces a `string_view` with a length greater than `max_size()`.

basic_string_view::operator=

Assigns a `string_view` or convertible string object to another `string_view`.

```
constexpr basic_string_view& operator=(const basic_string_view&) noexcept = default;
```

Example

```
string_view s = "Hello";  
string_view s2 = s;
```

basic_string_view::operator[]

Provides a `const`_reference to the character with a specified index.

```
constexpr const_reference operator[](size_type offset) const;
```

Parameters

offset

The index of the element to be referenced.

Return Value

A `const`_reference to the character at the position specified by the parameter index.

Remarks

The first element has an index of zero, and the following elements are indexed consecutively by the positive integers, so that a `string_view` of length n has an n th element indexed by the number $n - 1$.

`operator[]` is faster than the member function [at](#) for providing read access to the elements of a `string_view`.

`operator[]` does not check whether the index passed as an argument is valid. An invalid index passed to `operator[]` results in undefined behavior.

The reference returned may be invalidated if the underlying string data is modified or deleted by the owning object.

When compiling with `_ITERATOR_DEBUG_LEVEL` set to 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the `string_view`. For more information, see [Checked Iterators](#).

basic_string_view::rbegin

Returns a `const` iterator to the first element in a reversed `string_view`.

```
constexpr const_reverse_iterator rbegin() const noexcept;
```

Return Value

Returns a random-access iterator to the first element in a reversed `string_view`, addressing what would be the last element in the corresponding unreversed `string_view`.

Remarks

`rbegin` is used with a reversed `string_view` just as [begin](#) is used with a `string_view`. `rbegin` can be used to initialize an iteration backwards.

basic_string_view::remove_prefix

Moves the pointer forward by the specified number of elements.

```
constexpr void remove_prefix(size_type n);
```

Remarks

Leaves the underlying data unchanged. Moves the `string_view` pointer forward by `n` elements and sets the private `size` data member to `size - n`.

basic_string_view::remove_suffix

Reduces the size of the view by the specified number of elements starting from the back.

```
constexpr void remove_suffix(size_type n);
```

Remarks

Leaves the underlying data and the pointer to it unchanged. Sets the private `size` data member to `size - n`.

basic_string_view::rend

Returns a const iterator that points to one past the last element in a reversed `string_view`.

```
constexpr reverse_iterator rend() const noexcept;
```

Return Value

A const reverse random-access iterator that points to one past the last element in a reversed `string_view`.

Remarks

`rend` is used with a reversed `string_view` just as `end` is used with a `string_view`. `rend` can be used to test whether a reverse iterator has reached the end of its `string_view`. The value returned by `rend` should not be dereferenced.

basic_string_view::rfind

Searches a `string_view` in reverse for a substring that matches a specified sequence of characters.

```
constexpr size_type rfind(basic_string_view str, size_type offset = npos) const noexcept;  
constexpr size_type rfind(charT chVal, size_type offset = npos) const noexcept;  
constexpr size_type rfind(const charT* ptr, size_type offset, size_type count) const;  
constexpr size_type rfind(const charT* ptr, size_type offset = npos) const;
```

Parameters

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

str

The `string_view` for which the member function is to search.

Return Value

The index of the first character of the substring when successful; otherwise `npos`.

`basic_string_view::size`

Returns the number of elements in the `string_view`.

```
constexpr size_type size() const noexcept;
```

Return Value

The length of the `string_view`.

Remarks

A `string_view` can modify its length, for example by `remove_prefix` and `remove_suffix`. Because this does not modify the underlying string data, the size of a `string_view` is not necessarily the size of the underlying data.

`basic_string_view::substr`

Returns a `string_view` that represents (at most) the specified number of characters from a specified position.

```
constexpr basic_string_view substr(size_type offset = 0, size_type count = npos) const;
```

Parameters

offset

An index locating the element at the position from which the copy is made, with a default value of 0.

count

The number of characters to include in the substring, if they are present.

Return Value

A `string_view` object that represents the specified sub-sequence of elements.

`basic_string_view::swap`

Exchanges two `string_views`, in other words the pointers to the underlying string data, and the size values.

```
constexpr void swap(basic_string_view& sv) noexcept;
```

Parameters

sv

The source `string_view` whose pointer and size values are to be exchanged with that of the destination `string_view`.

See also

[<string_view>](#)

[Thread Safety in the C++ Standard Library](#)

hash<string_view> Specialization

4/24/2019 • 2 minutes to read • [Edit Online](#)

A template specialization that produces a hash value given a string_view.

```
template <class CharType, class Traits>
struct hash<basic_string_view<CharType, Traits>>
{
    typedef basic_string_view<CharType, Traits> argument_type;
    typedef size_t result_type;

    size_t operator()(const basic_string_view<CharType, Traits>) const
        noexcept;
};
```

Remarks

The hash of a string_view equals the hash of the underlying string object.

Example

```
//compile with: /std:c++17
#include <string>
#include <string_view>
#include <iostream>

using namespace std;

int main()
{
    string_view sv{ "Hello world" };
    string s{ "Hello world" };
    cout << boolalpha << (hash<string_view>{})(sv)
        == hash<string>{}(s)); // true
}
```

<strstream>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines several classes that support iostreams operations on sequences stored in an allocated array of **char** object. Such sequences are easily converted to and from C strings.

Syntax

```
#include <strstream>
```

Remarks

Objects of type `strstream` work with `char *`, which are C strings. Use `<sstream>` to work with objects of type `basic_string`.

NOTE

The classes in `<strstream>` are deprecated. Consider using the classes in `<sstream>` instead.

Classes

CLASS	DESCRIPTION
strstreambuf Class	The class describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a char array object.
istrstream Class	The class describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf .
ostrstream Class	The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class strstreambuf .
strstream Class	The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf .

See also

[<strstream>](#)

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

strstreambuf Class

11/8/2018 • 10 minutes to read • [Edit Online](#)

Describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a **char** array object.

Syntax

```
class strstreambuf : public streambuf
```

Remarks

Depending on how the object is constructed, it can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `strstreambuf` stores several bits of mode information as its `strstreambuf` mode. These bits indicate whether the controlled sequence:

- Has been allocated and needs to be freed eventually.
- Is modifiable.
- Is extendable by reallocating storage.
- Has been frozen and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object.

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control `strstreambuf` allocation. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

NOTE

This class is deprecated. Consider using [stringbuf](#) or [wstringbuf](#) instead.

Constructors

CONSTRUCTOR	DESCRIPTION
strstreambuf	Constructs an object of type <code>strstreambuf</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
freeze	Causes a stream buffer to be unavailable through stream buffer operations.

MEMBER FUNCTION	DESCRIPTION
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	A protected virtual member function that tries to put back an element into the input stream, and then make it the current element (pointed to by the next pointer).
pcount	Returns a count of the number of elements written to the controlled sequence.
seekoff	A protected virtual member function that tries to alter the current positions for the controlled streams.
seekpos	A protected virtual member function that tries to alter the current positions for the controlled streams.
str	Calls freeze , and then returns a pointer to the beginning of the controlled sequence.
underflow	A protected virtual function to extract the current element from the input stream.

Requirements

Header: <strstream>

Namespace: std

strstreambuf::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

If *_Freezeit* is true, the function alters the stored `strstreambuf` mode to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

[str](#) implies `freeze`.

NOTE

A frozen buffer will not be freed during `strstreambuf` destruction. You must unfreeze the buffer before it is freed to avoid a memory leak.

Example

```

// strstreambuf_freeze.cpp
// compile with: /EHsc

#include <iostream>
#include <strstream>

using namespace std;

void report(strstream &x)
{
    if (!x.good())
        cout << "stream bad" << endl;
    else
        cout << "stream good" << endl;
}

int main()
{
    strstream x;

    x << "test1";
    cout << "before freeze: ";
    report(x);

    // Calling str freezes stream.
    cout.write(x.rdbuf()->str(), 5) << endl;
    cout << "after freeze: ";
    report(x);

    // Stream is bad now, wrote on frozen stream
    x << "test1.5";
    cout << "after write to frozen stream: ";
    report(x);

    // Unfreeze stream, but it is still bad
    x.rdbuf()->freeze(false);
    cout << "after unfreezing stream: ";
    report(x);

    // Clear stream
    x.clear();
    cout << "after clearing stream: ";
    report(x);

    x << "test3";
    cout.write(x.rdbuf()->str(), 10) << endl;

    // Clean up. Failure to unfreeze stream will cause a
    // memory leak.
    x.rdbuf()->freeze(false);
}

```

```

before freeze: stream good
test1
after freeze: stream good
after write to frozen stream: stream bad
after unfreezing stream: stream bad
after clearing stream: stream good
test1test3

```

strstreambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
virtual int overflow(int _Meta = EOF);
```

Parameters

_Meta

The character to insert into the buffer, or `EOF`.

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, if *_Meta* == `EOF`, it returns some value other than `EOF`. Otherwise, it returns *_Meta*.

Remarks

If *_Meta* != `EOF`, the protected virtual member function tries to insert the element `(char)_Meta` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored `strstreambuf` mode says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. Extending the output buffer this way also extends any associated input buffer.

`strstreambuf::pbackfail`

A protected virtual member function that tries to put back an element into the input stream, and then makes it the current element (pointed to by the next pointer).

```
virtual int pbackfail(int _Meta = EOF);
```

Parameters

_Meta

The character to insert into the buffer, or `EOF`.

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, if *_Meta* == `EOF`, it returns some value other than `EOF`. Otherwise, it returns *_Meta*.

Remarks

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

If *_Meta* == `EOF`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `ch = (char)_Meta`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `ch`, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the `strstreambuf` mode says the controlled sequence is modifiable, the function can store `ch` into the putback position and decrement the next pointer for the input buffer.

`strstreambuf::pcount`

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount() const;
```

Return Value

A count of the number of elements written to the controlled sequence.

Remarks

Specifically, if `pptr` is a null pointer, the function returns zero. Otherwise, it returns `pptr` - `pbase`.

Example

```
// strstreambuf_pcount.cpp
// compile with: /EHsc
#include <iostream>
#include <strstream>
using namespace std;

int main( )
{
    strstream x;
    x << "test1";
    cout << x.rdbuf( )->pcount( ) << endl;
    x << "test2";
    cout << x.rdbuf( )->pcount( ) << endl;
}
```

strstreambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual streampos seekoff(streamoff _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off

The position to seek for relative to *_Way*.

_Way

The starting point for offset operations. See [seekdir](#) for possible values.

_Which

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `_Way == ios_base::beg`, the new position is the beginning of the stream plus *_Off*.

- If `_Way == ios_base::cur`, the new position is the current stream position plus `_Off`.
- If `_Way == ios_base::end`, the new position is the end of the stream plus `_Off`.

If `_Which & ios_base::in` is nonzero and the input buffer exist, the function alters the next position to read in the input buffer. If `_Which & ios_base::out` is also nonzero, `_Way != ios_base::cur`, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if `_Which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

stringstream::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
virtual streampos seekpos(streampos _Sp, ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

`_Sp`

The position to seek for.

`_Which`

Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `stringstream`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `_Sp`.

If `_Which & ios_base::in` is nonzero and the input buffer exists, the function alters the next position to read in the input buffer. If `_Which & ios_base::out` is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read. Otherwise, if `_Which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

stringstream::str

Calls [freeze](#), and then returns a pointer to the beginning of the controlled sequence.

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

No terminating null element exists, unless you explicitly insert one.

Example

See [strstreambuf::freeze](#) for a sample that uses **str**.

strstreambuf::strstreambuf

Constructs an object of type `strstreambuf`.

```
explicit strstreambuf(streamsize count = 0);

strstreambuf(void (* _Allocfunc)(size_t),
             void (* _Freefunc)(void*));

strstreambuf(char* _Getptr,
             streamsize count,
             char* _Putptr = 0);

strstreambuf(signed char* _Getptr,
             streamsize count,
             signed char* _Putptr = 0);

strstreambuf(unsigned char* _Getptr,
             streamsize count,
             unsigned char* _Putptr = 0);

strstreambuf(const char* _Getptr,
             streamsize count);

strstreambuf(const signed char* _Getptr,
             streamsize count);

strstreambuf(const unsigned char* _Getptr,
             streamsize count);
```

Parameters

_Allocfunc

The function used to allocate buffer memory.

count

Determines the length of the buffer pointed to by *_Getptr*. If *_Getptr* is not an argument (first constructor form), a suggested allocation size for the buffers.

_Freefunc

The function used to free buffer memory.

_Getptr

A buffer used for input.

_Putptr

A buffer used for output.

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer, the output buffer, and `strstreambuf` allocation. It sets the stored `strstreambuf` mode to make the controlled sequence modifiable and extendable. It also accepts *count* as a suggested initial allocation size.

The second constructor behaves like the first, except that it stores *_Allocfunc* as the pointer to the function to call to allocate storage and *_Freefunc* as the pointer to the function to call to free that storage.

The three constructors:

```

strstreambuf(char *_Getptr,
             streamsize count,
             char *putptr = 0);

strstreambuf(signed char *_Getptr,
             streamsize count,
             signed char *putptr = 0);

strstreambuf(unsigned char *_Getptr,
             streamsize count,
             unsigned char *putptr = 0);

```

also behave like the first, except that `_Getptr` designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements N in the array is determined as follows:

- If `(count > 0)`, then N is `count`.
- If `(count == 0)`, then N is `strlen((const char *)_Getptr)`.
- If `(count < 0)`, then N is **INT_MAX**.

If `_Putptr` is a null pointer, the function establishes just an input buffer by executing:

```

setg(_Getptr,
     _Getptr,
     _Getptr + N);

```

Otherwise, it establishes both input and output buffers by executing:

```

setg(_Getptr,
     _Getptr,
     _Putptr);

setp(_Putptr,
     _Getptr + N);

```

In this case, `_Putptr` must be in the interval `[_Getptr, _Getptr + N]`.

Finally, the three constructors:

```

strstreambuf(const char *_Getptr,
             streamsize count);

strstreambuf(const signed char *_Getptr,
             streamsize count);

strstreambuf(const unsigned char *_Getptr,
             streamsize count);

```

all behave the same as:

```

streambuf((char *)_Getptr, count);

```

except that the stored mode makes the controlled sequence neither modifiable nor extendable.

strstreambuf::underflow

A protected virtual function to extract the current element from the input stream.

```
virtual int underflow();
```

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, it returns the current element in the input stream, converted as described above.

Remarks

The protected virtual member function endeavors to extract the current element `ch` from the input buffer, then advance the current stream position, and return the element as `(int)(unsigned char)ch`. It can do so in only one way: if a read position is available, it takes `ch` as the element stored in the read position and advances the next pointer for the input buffer.

See also

[streambuf](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

istream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class `strstreambuf`.

Syntax

```
class istream : public istream
```

Remarks

The object stores an object of class `strstreambuf`.

NOTE

This class is deprecated. Consider using `istream` or `wistream` instead.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>istream</code>	Constructs an object of type <code>istream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>rdbuf</code>	Returns a pointer to the stream's associated <code>strstreambuf</code> object.
<code>str</code>	Calls <code>freeze</code> , and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: `<istream>`

Namespace: `std`

`istream::istream`

Constructs an object of type `istream`.

```
explicit istrstream(
    const char* ptr);

explicit istrstream(
    char* ptr);

istrstream(
    const char* ptr,
    streamsize count);

istrstream(
    char* ptr,
    int count);
```

Parameters

count

The length of the buffer (*ptr*).

ptr

The contents with which the buffer is initialized.

Remarks

All the constructors initialize the base class by calling [istream\(sb\)](#), where `sb` is the stored object of class [strstreambuf](#). The first two constructors also initialize `sb` by calling `strstreambuf((const char *) ptr, 0)`. The remaining two constructors instead call `strstreambuf((const char *) ptr, count)`.

istrstream::rdbuf

Returns a pointer to the stream's associated [strstreambuf](#) object.

```
strstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated [strstreambuf](#) object.

Remarks

The member function returns the address of the stored stream buffer, of type pointer to [strstreambuf](#).

Example

See [strstreambuf::pcount](#) for a sample that uses `rdbuf`.

istrstream::str

Calls [freeze](#), and then returns a pointer to the beginning of the controlled sequence.

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns `rdbuf` -> `str`.

Example

See [istream::str](#) for a sample that uses `str`.

See also

[istream](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

ostream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class `strstreambuf`.

Syntax

```
class ostream : public ostream
```

Remarks

The object stores an object of class `strstreambuf`.

NOTE

This class is deprecated. Consider using `ostream` or `wostringstream` instead.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>ostream</code>	Constructs an object of type <code>ostream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>freeze</code>	Causes a stream buffer to be unavailable through stream buffer operations.
<code>pcount</code>	Returns a count of the number of elements written to the controlled sequence.
<code>rdbuf</code>	Returns a pointer to the stream's associated <code>strstreambuf</code> object.
<code>str</code>	Calls <code>freeze</code> , and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: `<ostream>`

Namespace: `std`

`ostream::freeze`

Causes a stream buffer to be unavailable through stream buffer operations.

```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

The member function calls `rdbuf -> freeze(_Freezeit)`.

Example

See `stringstream::freeze` for an example that uses `freeze`.

ostream::ostream

Constructs an object of type `ostream`.

```
ostream();

ostream(char* ptr,
        streamsize count,
        ios_base::openmode _Mode = ios_base::out);
```

Parameters

ptr

The buffer.

count

The size of the buffer in bytes.

_Mode

The input and output mode of the buffer. See `ios_base::openmode` for more information.

Remarks

Both constructors initialize the base class by calling `ostream(sb)`, where `sb` is the stored object of class `stringstreambuf`. The first constructor also initializes `sb` by calling `stringstreambuf`. The second constructor initializes the base class one of two ways:

- If `_Mode & ios_base::app == 0`, then `ptr` must designate the first element of an array of `count` elements, and the constructor calls `stringstreambuf(ptr, count, ptr)`.
- Otherwise, `ptr` must designate the first element of an array of count elements that contains a C string whose first element is designated by `ptr`, and the constructor calls `stringstreambuf(ptr, count, ptr + strlen(ptr))`.

ostream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount() const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns [rdbuf](#) -> [pcount](#).

Example

See [stringstream::pcount](#) for a sample that uses `pcount`.

stringstream::rdbuf

Returns a pointer to the stream's associated stringstreambuf object.

```
stringstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated stringstreambuf object.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to [stringstreambuf](#).

Example

See [stringstreambuf::pcount](#) for a sample that uses `rdbuf`.

stringstream::str

Calls [freeze](#), and then returns a pointer to the beginning of the controlled sequence.

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns [rdbuf](#) -> [str](#).

Example

See [stringstream::str](#) for a sample that uses `str`.

See also

[ostream](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

stringstream Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class [stringstreambuf](#).

Syntax

```
class stringstream : public istream
```

Remarks

The object stores an object of class `stringstreambuf`.

NOTE

This class is deprecated. Consider using [stringstream](#) or [wstringstream](#) instead.

Constructors

CONSTRUCTOR	DESCRIPTION
stringstream	Constructs an object of type <code>stringstream</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
rdbuf	Returns a pointer to the stream's associated <code>stringstreambuf</code> object.
str	Calls freeze , and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: <stringstream>

Namespace: std

stringstream::freeze

Causes a stream buffer to be unavailable through stream buffer operations.


```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit

A **bool** indicating whether you want the stream to be frozen.

Remarks

The member function calls `rdbuf -> freeze(_Freezeit)`.

Example

See `stringstream::freeze` for an example that uses `freeze`.

stringstream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
streamsize pcount() const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns `rdbuf -> pcount`.

Example

See `stringstream::pcount` for a sample of using `pcount`.

stringstream::rdbuf

Returns a pointer to the stream's associated `stringstreambuf` object.

```
stringstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated `stringstreambuf` object.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to `stringstreambuf`.

Example

See `stringstreambuf::pcount` for a sample that uses `rdbuf`.

stringstream::str

Calls `freeze`, and then returns a pointer to the beginning of the controlled sequence.

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns `rdbuf` -> `str`.

Example

See `stringstream::str` for a sample that uses `str`.

stringstream::stringstream

Constructs an object of type `stringstream`.

```
stringstream();

stringstream(char* ptr,
             streamsize count,
             ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

count

The size of the buffer.

_Mode

The input and output mode of the buffer. See `ios_base::openmode` for more information.

ptr

The buffer.

Remarks

Both constructors initialize the base class by calling `streambuf(sb)`, where `sb` is the stored object of class `stringstream`. The first constructor also initializes `sb` by calling `stringstream`. The second constructor initializes the base class one of two ways:

- If `_Mode` & `ios_base::app` == 0, then *ptr* must designate the first element of an array of `count` elements, and the constructor calls `stringstream(ptr, count, ptr)`.
- Otherwise, *ptr* must designate the first element of an array of count elements that contains a C string whose first element is designated by *ptr*, and the constructor calls `stringstream(ptr, count, ptr + strlen(ptr))`.

See also

[iostream](#)

[Thread Safety in the C++ Standard Library](#)

[iostream Programming](#)

[iostreams Conventions](#)

<system_error>

3/18/2019 • 2 minutes to read • [Edit Online](#)

Include the header `<system_error>` to define the exception class `system_error` and related templates for processing low-level system errors.

Syntax

```
#include <system_error>
```

Objects

<code>generic_category</code>	Represents the category for generic errors.
<code>system_category</code>	Represents the category for errors caused by low-level system overflows.

Functions

FUNCTION	DESCRIPTION
<code>make_error_code</code>	Creates an <code>error_code</code> object.
<code>make_error_condition</code>	Creates an <code>error_condition</code> object.

Operators

OPERATOR	DESCRIPTION
<code>operator==</code>	Tests if the object on the left side of the operator is equal to the object on the right side.
<code>operator!=</code>	Tests if the object on the left side of the operator is not equal to the object on the right side.
<code>operator<</code>	Tests if an object is less than the object passed in for comparison.

Enumerations

<code>errc</code>	Provides symbolic names for all the error-code macros defined by Posix in <code><errno.h></code> .

Classes and Structs

--	--

error_category	Represents the abstract, common base for objects that describes a category of error codes.
error_code	Represents low-level system errors that are implementation-specific.
error_condition	Represents user-defined error codes.
is_error_code_enum	Represents a type predicate that tests for the error_code Class enumeration.
is_error_condition_enum	Represents a type predicate that tests for the error_condition Class enumeration.
system_error	Represents the base class for all exceptions thrown to report a low-level system overflow.

Requirements

Header: `<system_error>`

Namespace: `std`

See also

[Header Files Reference](#)

<system_error> functions

3/18/2019 • 2 minutes to read • [Edit Online](#)

generic_category	make_error_code	make_error_condition
system_category		

generic_category

Represents the category for generic errors.

```
const error_category& generic_category() noexcept;
```

Remarks

The `generic_category` object is an implementation of [error_category](#).

make_error_code

Creates an error code object.

```
error_code make_error_code(std::errc error) noexcept;
```

Parameters

error

The `std::errc` enumeration value to store in the error code object.

Return Value

The error code object.

Remarks

make_error_condition

Creates an error condition object.

```
error_condition make_error_condition(std::errc error) noexcept;
```

Parameters

error

The `std::errc` enumeration value to store in the error code object.

Return Value

The error condition object.

Remarks

system_category

Represents the category for errors caused by low-level system overflows.

```
const error_category& system_category() noexcept;
```

Remarks

The `system_category` object is an implementation of [error_category](#).

See also

[<system_error>](#)

<system_error> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

operator!=

operator<

operator==

operator==

Tests if the object on the left side of the operator is equal to the object on the right side.

```
bool operator==(const error_code& left,
                const error_condition& right);

bool operator==(const error_condition& left,
                const error_code& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	The object to be tested for equality.
<i>right</i>	The object to be tested for equality.

Return Value

true if the objects are equal; **false** if objects are not equal.

Remarks

This function returns `left.category() == right.category() && left.value() == right.value()`.

operator!=

Tests if the object on the left side of the operator is not equal to the object on the right side.

```
bool operator!=(const error_code& left,
                const error_condition& right);

bool operator!=(const error_condition& left,
                const error_code& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	The object to be tested for inequality.
<i>right</i>	The object to be tested for inequality.

Return Value

true if the object passed in *left* is not equal to the object passed in *right*; otherwise **false**.

Remarks

This function returns `!(left == right)` .

operator<

Tests if an object is less than the object passed in for comparison.

```
template <class _Enum>
inline bool operator<(
    _Enum left,
    typename enable_if<is_error_code_enum<_Enum>::value,
    const error_code&>::type right);

template <class _Enum>
inline bool operator<(
    typename enable_if<is_error_code_enum<_Enum>::value,
    const error_code&>::type left, _Enum right);

template <class _Enum>
inline bool operator<(
    _Enum left,
    typename enable_if<is_error_condition_enum<_Enum>::value,
    const error_condition&>::type right);

template <class _Enum>
inline bool operator<(
    typename enable_if<is_error_condition_enum<_Enum>::value,
    const error_condition&>::type left, _Enum right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	The object to be compared.
<i>right</i>	The object to be compared.

Return Value

true if the object passed in *left* is less than the object passed in *right*; Otherwise, **false**.

Remarks

This function tests the error order.

See also

[<system_error>](#)

<system_error> enums

10/31/2018 • 2 minutes to read • [Edit Online](#)

errc	io_errc

errc Enumeration

Provides symbolic names for all the error-code macros defined by Posix in `<errno.h>`.

```
class errc { address_family_not_supported = EAFNOSUPPORT, address_in_use = EADDRINUSE,
address_not_available = EADDRNOTAVAIL, already_connected = EISCONN, argument_list_too_long = E2BIG,
argument_out_of_domain = EDOM, bad_address = EFAULT, bad_file_descriptor = EBADF, bad_message =
EBADMSG, broken_pipe = EPIPE, connection_aborted = ECONNABORTED, connection_already_in_progress =
EALREADY, connection_refused = ECONNREFUSED, connection_reset = ECONNRESET, cross_device_link =
EXDEV, destination_address_required = EDESTADDRREQ, device_or_resource_busy = EBUSY,
directory_not_empty = ENOTEMPTY, executable_format_error = ENOEXEC, file_exists = EEXIST, file_too_large =
EFBIG, filename_too_long = ENAMETOOLONG, function_not_supported = ENOSYS, host_unreachable =
EHOSTUNREACH, identifier_removed = EIDRM, illegal_byte_sequence = EILSEQ,
inappropriate_io_control_operation = ENOTTY, interrupted = EINTR, invalid_argument = EINVAL, invalid_seek =
ESPIPE, io_error = EIO, is_a_directory = EISDIR, message_size = EMSGSIZE, network_down = ENETDOWN,
network_reset = ENETRESET, network_unreachable = ENETUNREACH, no_buffer_space = ENOBUFS,
no_child_process = ECHILD, no_link = ENOLINK, no_lock_available = ENOLCK, no_message_available =
ENODATA, no_message = ENOMSG, no_protocol_option = ENOPROTOOPT, no_space_on_device = ENOSPC,
no_stream_resources = ENOSR, no_such_device_or_address = ENXIO, no_such_device = ENODEV,
no_such_file_or_directory = ENOENT, no_such_process = ESRCH, not_a_directory = ENOTDIR, not_a_socket =
ENOTSOCK, not_a_stream = ENOSTR, not_connected = ENOTCONN, not_enough_memory = ENOMEM,
not_supported = ENOTSUP, operation_canceled = ECANCELED, operation_in_progress = EINPROGRESS,
operation_not_permitted = EPERM, operation_not_supported = EOPNOTSUPP, operation_would_block =
EWOULDBLOCK, owner_dead = EOWNERDEAD, permission_denied = EACCES, protocol_error = EPROTO,
protocol_not_supported = EPROTONOSUPPORT, read_only_file_system = EROFS,
resource_deadlock_would_occur = EDEADLK, resource_unavailable_try_again = EAGAIN, result_out_of_range =
ERANGE, state_not_recoverable = ENOTRECOVERABLE, stream_timeout = ETIME, text_file_busy = ETXTBSY,
timed_out = ETIMEDOUT, too_many_files_open_in_system = ENFILE, too_many_files_open = EMFILE,
too_many_links = EMLINK, too_many_symbolic_link_levels = ELOOP, value_too_large = EOVERFLOW,
wrong_protocol_type = EPROTOTYPE,};
```

Remarks

io_errc Enumeration

Provides symbolic names for the error conditions in `<iostream>`. Can be used to create [error_condition](#) objects to be compared with the value that's returned by the [ios_base::failure](#) `code()` function.

```
class io_errc { stream = 1 };
```

Remarks

Both [std::make_error_code\(\)](#) and [std::make_error_condition\(\)](#) are overloaded for this enum.

[ios_base::failure](#) can contain categories of error codes other than [error_condition](#).

Example

```
// io_errc.cpp
// cl.exe /nologo /W4 /EHsc /MTd

#include <iostream>

using namespace std;

int main()
{
    cin.exceptions(ios::failbit | ios::badbit);

    try {
        cin.rdbuf(nullptr); // throws io_errc::stream
    }
    catch (ios::failure& e) {
        cerr << "ios failure caught: ";
        if (e.code() == make_error_condition(io_errc::stream)) {
            cerr << "io_errc stream error condition" << endl;
        }
        else {
            cerr << "unmatched error condition code " << e.code() << endl;
        }
    }
}
```

See also

[<system_error>](#)

error_category Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents the abstract, common base for objects that describes a category of error codes.

Syntax

```
class error_category;
```

Remarks

Two predefined objects implement `error_category`: [generic_category](#) and [system_category](#).

Typedefs

TYPE NAME	DESCRIPTION
value_type	A type that represents the stored error code value.

Member functions

MEMBER FUNCTION	DESCRIPTION
default_error_condition	Stores the error code value for an error condition object.
equivalent	Returns a value that specifies whether error objects are equivalent.
message	Returns the name of the specified error code.
name	Returns the name of the category.

Operators

OPERATOR	DESCRIPTION
operator==	Tests for equality between <code>error_category</code> objects.
operator!=	Tests for inequality between <code>error_category</code> objects.
operator<	Tests if the error_category object is less than the <code>error_category</code> object passed in for comparison.

Requirements

Header: <system_error>

Namespace: std

error_category::default_error_condition

Stores the error code value for an error condition object.

```
virtual error_condition default_error_condition(int _Errval) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>_Errval</i>	The error code value to store in the error_condition .

Return Value

Returns `error_condition(_Errval, *this)`.

Remarks

error_category::equivalent

Returns a value that specifies whether error objects are equivalent.

```
virtual bool equivalent(value_type _Errval,  
    const error_condition& _Cond) const;  
  
virtual bool equivalent(const error_code& _Code,  
    value_type _Errval) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>_Errval</i>	The error code value to compare.
<i>_Cond</i>	The error_condition object to compare.
<i>_Code</i>	The error_code object to compare.

Return Value

true if the category and value are equal; otherwise, **false**.

Remarks

The first member function returns `*this == _Cond.category() && _Cond.value() == _Errval`.

The second member function returns `*this == _Code.category() && _Code.value() == _Errval`.

error_category::message

Returns the name of the specified error code.

```
virtual string message(error_code::value_type val) const = 0;
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The error code value to describe.

Return Value

Returns a descriptive name of the error code *val* for the category.

Remarks

error_category::name

Returns the name of the category.

```
virtual const char *name() const = 0;
```

Return Value

Returns the name of the category as a null-terminated byte string.

Remarks

error_category::operator==

Tests for equality between `error_category` objects.

```
bool operator==(const error_category& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for equality.

Return Value

true if the objects are equal; **false** if the objects are not equal.

Remarks

This member operator returns `this == &right`.

error_category::operator!=

Tests for inequality between `error_category` objects.

```
bool operator!=(const error_category& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for inequality.

Return Value

true if the `error_category` object is not equal to the `error_category` object passed in *right*; otherwise **false**.

Remarks

The member operator returns `(!*this == right)`.

error_category::operator<

Tests if the [error_category](#) object is less than the `error_category` object passed in for comparison.

```
bool operator<(const error_category& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>error_category</code> object to be compared.

Return Value

true if the `error_category` object is less than the `error_category` object passed in for comparison; Otherwise, **false**.

Remarks

The member operator returns `this < &right`.

error_category::value_type

A type that represents the stored error code value.

```
typedef int value_type;
```

Remarks

This type definition is a synonym for **int**.

See also

[<system_error>](#)

error_code Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Represents low-level system errors that are implementation-specific.

Syntax

```
class error_code;
```

Remarks

An object of type `error_code` class stores an error code value and a pointer to an object that represents a [category](#) of error codes that describe reported low-level system errors.

Constructors

CONSTRUCTOR	DESCRIPTION
error_code	Constructs an object of type <code>error_code</code> .

Typedefs

TYPE NAME	DESCRIPTION
value_type	A type that represents the stored error code value.

Member functions

MEMBER FUNCTION	DESCRIPTION
assign	Assigns an error code value and category to an error code.
category	Returns the error category.
clear	Clears the error code value and category.
default_error_condition	Returns the default error condition.
message	Returns the name of the error code.

Operators

OPERATOR	DESCRIPTION
operator==	Tests for equality between <code>error_code</code> objects.
operator!=	Tests for inequality between <code>error_code</code> objects.

OPERATOR	DESCRIPTION
<code>operator<</code>	Tests if the <code>error_code</code> object is less than the <code>error_code</code> object passed in for comparison.
<code>operator=</code>	Assigns a new enumeration value to the <code>error_code</code> object.
<code>operator bool</code>	Casts a variable of type <code>error_code</code> .

Requirements

Header: `<system_error>`

Namespace: `std`

`error_code::assign`

Assigns an error code value and category to an error code.

```
void assign(value_type val, const error_category& _Cat);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The error code value to store in the <code>error_code</code> .
<i>_Cat</i>	The error category to store in the <code>error_code</code> .

Remarks

The member function stores *val* as the error code value and a pointer to *_Cat*.

`error_code::category`

Returns the error category.

```
const error_category& category() const;
```

Remarks

`error_code::clear`

Clears the error code value and category.

```
clear();
```

Remarks

The member function stores a zero error code value and a pointer to the [generic_category](#) object.

`error_code::default_error_condition`

Returns the default error condition.

```
error_condition default_error_condition() const;
```

Return Value

The [error_condition](#) specified by [default_error_condition](#).

Remarks

This member function returns `category().default_error_condition(value())`.

error_code::error_code

Constructs an object of type `error_code`.

```
error_code();

error_code(value_type val, const error_category& _Cat);

template <class _Enum>
error_code(_Enum _Errcode,
           typename enable_if<is_error_code_enum<_Enum>::value,
                               error_code::type* = 0>);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The error code value to store in the <code>error_code</code> .
<i>_Cat</i>	The error category to store in the <code>error_code</code> .
<i>_Errcode</i>	The enumeration value to store in the <code>error_code</code> .

Remarks

The first constructor stores a zero error code value and a pointer to the [generic_category](#).

The second constructor stores *val* as the error code value and a pointer to [error_category](#).

The third constructor stores `(value_type)_Errcode` as the error code value and a pointer to the [generic_category](#).

error_code::message

Returns the name of the error code.

```
string message() const;
```

Return Value

A `string` representing the name of the error code.

Remarks

This member function returns `category().message(value())`.

error_code::operator==

Tests for equality between `error_code` objects.

```
bool operator==(const error_code& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for equality.

Return Value

true if the objects are equal; **false** if objects are not equal.

Remarks

The member operator returns `category() == right.category() && value == right.value()`.

error_code::operator!=

Tests for inequality between `error_code` objects.

```
bool operator!=(const error_code& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for inequality.

Return Value

true if the `error_code` object is not equal to the `error_code` object passed in *right*; otherwise **false**.

Remarks

The member operator returns `!(*this == right)`.

error_code::operator<

Tests if the `error_code` object is less than the `error_code` object passed in for comparison.

```
bool operator<(const error_code& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>error_code</code> object to be compared.

Return Value

true if the `error_code` object is less than the `error_code` object passed in for comparison; Otherwise, **false**.

Remarks

The member operator returns

```
category() < right.category() || category() == right.category() && value < right.value()
```

error_code::operator=

Assigns a new enumeration value to the `error_code` object.

```
template <class _Enum>
typename enable_if<is_error_code_enum<_Enum>::value, error_code>::type&
operator=(_Enum _Errcode);
```

Parameters

PARAMETER	DESCRIPTION
<code>_Errcode</code>	The enumeration value to assign to the <code>error_code</code> object.

Return Value

A reference to the `error_code` object that is being assigned the new enumeration value by the member function.

Remarks

The member operator stores `(value_type)_Errcode` as the error code value and a pointer to the [generic_category](#). It returns `*this`.

error_code::operator bool

Casts a variable of type `error_code`.

```
explicit operator bool() const;
```

Return Value

The Boolean value of the `error_code` object.

Remarks

The operator returns a value convertible to **true** only if [value](#) is not equal to zero. The return type is convertible only to **bool**, not to `void *` or other known scalar types.

error_code::value

Returns the stored error code value.

```
value_type value() const;
```

Return Value

The stored error code value of type [value_type](#).

Remarks

error_code::value_type

A type that represents the stored error code value.

```
typedef int value_type;
```

Remarks

This type definition is a synonym for **int**.

See also

[error_category](#) Class

[<system_error>](#)

error_condition Class

10/31/2018 • 3 minutes to read • [Edit Online](#)

Represents user-defined error codes.

Syntax

```
class error_condition;
```

Remarks

An object of type `error_condition` stores an error code value and a pointer to an object that represents a [category](#) of error codes used for reported user-defined errors.

Constructors

CONSTRUCTOR	DESCRIPTION
error_condition	Constructs an object of type <code>error_condition</code> .

Typedefs

TYPE NAME	DESCRIPTION
value_type	A type that represents the stored error code value.

Member functions

MEMBER FUNCTION	DESCRIPTION
assign	Assigns an error code value and category to an error condition.
category	Returns the error category.
clear	Clears the error code value and category.
message	Returns the name of the error code.

Operators

OPERATOR	DESCRIPTION
operator==	Tests for equality between <code>error_condition</code> objects.
operator!=	Tests for inequality between <code>error_condition</code> objects.
operator<	Tests if the <code>error_condition</code> object is less than the <code>error_code</code> object passed in for comparison.

OPERATOR	DESCRIPTION
<code>operator=</code>	Assigns a new enumeration value to the <code>error_condition</code> object.
<code>operator bool</code>	Casts a variable of type <code>error_condition</code> .

Requirements

Header: `<system_error>`

Namespace: `std`

`error_condition::assign`

Assigns an error code value and category to an error condition.

```
void assign(value_type val, const error_category& _Cat);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The error code value to store in the <code>error_code</code> .
<i>_Cat</i>	The error category to store in the <code>error_code</code> .

Remarks

The member function stores *val* as the error code value and a pointer to *_Cat*.

`error_condition::category`

Returns the error category.

```
const error_category& category() const;
```

Return Value

A reference to the stored error category

Remarks

`error_condition::clear`

Clears the error code value and category.

```
clear();
```

Remarks

The member function stores a zero error code value and a pointer to the `generic_category` object.

`error_condition::error_condition`

Constructs an object of type `error_condition` .

```
error_condition();

error_condition(value_type val, const error_category& _Cat);

template <class _Enum>
error_condition(_Enum _Errcode,
               typename enable_if<is_error_condition_enum<_Enum>::value,
               error_code>::type* = 0);
```

Parameters

PARAMETER	DESCRIPTION
<i>val</i>	The error code value to store in the <code>error_condition</code> .
<i>_Cat</i>	The error category to store in the <code>error_condition</code> .
<i>_Errcode</i>	The enumeration value to store in the <code>error_condition</code> .

Remarks

The first constructor stores a zero error code value and a pointer to the [generic_category](#).

The second constructor stores *val* as the error code value and a pointer to [error_category](#).

The third constructor stores `(value_type)_Errcode` as the error code value and a pointer to the [generic_category](#).

error_condition::message

Returns the name of the error code.

```
string message() const;
```

Return Value

A `string` representing the name of the error code.

Remarks

This member function returns `category().message(value())` .

error_condition::operator==

Tests for equality between `error_condition` objects.

```
bool operator==(const error_condition& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for equality.

Return Value

true if the objects are equal; **false** if objects are not equal.

Remarks

The member operator returns `category() == right.category() && value == right.value()` .

error_condition::operator!=

Tests for inequality between `error_condition` objects.

```
bool operator!=(const error_condition& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The object to be tested for inequality.

Return Value

true if the `error_condition` object is not equal to the `error_condition` object passed in *right*; otherwise **false**.

Remarks

The member operator returns `!(*this == right)` .

error_condition::operator<

Tests if the `error_condition` object is less than the `error_code` object passed in for comparison.

```
bool operator<(const error_condition& right) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>error_condition</code> object to be compared.

Return Value

true if the `error_condition` object is less than the `error_condition` object passed in for comparison; Otherwise, **false**.

Remarks

The member operator returns

```
category() < right.category() || category() == right.category() && value < right.value()
```

 .

error_condition::operator=

Assigns a new enumeration value to the `error_condition` object.

```
template <class _Enum>
error_condition(_Enum error,
    typename enable_if<is_error_condition_enum<_Enum>::value,
    error_condition>::type&
    operator=(Enum _Errcode);
```


Parameters

PARAMETER	DESCRIPTION
<code>_Errcode</code>	The enumeration value to assign to the <code>error_condition</code> object.

Return Value

A reference to the `error_condition` object that is being assigned the new enumeration value by the member function.

Remarks

The member operator stores `(value_type)error` as the error code value and a pointer to the [generic_category](#). It returns `*this`.

error_condition::operator bool

Casts a variable of type `error_condition`.

```
explicit operator bool() const;
```

Return Value

The Boolean value of the `error_condition` object.

Remarks

The operator returns a value convertible to **true** only if [value](#) is not equal to zero. The return type is convertible only to **bool**, not to `void *` or other known scalar types.

error_condition::value

Returns the stored error code value.

```
value_type value() const;
```

Return Value

The stored error code value of type [value_type](#).

Remarks

error_condition::value_type

A type that represents the stored error code value.

```
typedef int value_type;
```

Remarks

The type definition is a synonym for **int**.

See also

[error_category Class](#)

[<system_error>](#)

is_error_code_enum Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a type predicate that tests for the [error_code](#) enumeration.

Syntax

```
template <_Enum>
class is_error_code_enum;
```

Remarks

An instance of this [type predicate](#) holds true if the type `_Enum` is an enumeration value suitable for storing in an object of type `error_code`.

It is permissible to add specializations to this type for user-defined types.

Requirements

Header: `<system_error>`

Namespace: `std`

See also

[<type_traits>](#)

[<system_error>](#)

is_error_condition_enum Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents a type predicate that tests for the [error_condition](#) enumeration.

Syntax

```
template <_Enum>
class is_error_condition_enum;
```

Remarks

An instance of this [type predicate](#) holds true if the type `_Enum` is an enumeration value suitable for storing in an object of type `error_condition`.

It is permissible to add specializations to this type for user-defined types.

Requirements

Header: `<system_error>`

Namespace: `std`

See also

[<type_traits>](#)

[<system_error>](#)

system_error Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Represents the base class for all exceptions thrown to report a low-level system error.

Syntax

```
class system_error : public runtime_error {
public:
    explicit system_error(error_code _Errcode,
        const string& _Message = "");

    system_error(error_code _Errcode,
        const char *_Message);

    system_error(error_code::value_type _Errval,
        const error_category& _Errcat,
        const string& _Message);

    system_error(error_code::value_type _Errval,
        const error_category& _Errcat,
        const char *_Message);
    const error_code& code() const throw();
    const error_code& code() const throw();

};
```

Remarks

The value returned by `what` in the class `exception` is constructed from `_Message` and the stored object of type `error_code` (either `code` or `error_code(_Errval, _Errcat)`).

The member function `code` returns the stored `error_code` object.

Requirements

Header: <system_error>

Namespace: std

See also

<system_error>

<thread>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Include the standard header `<thread>` to define the class **thread** and various supporting functions.

Syntax

```
#include <thread>
```

Remarks

NOTE

In code that is compiled by using `/clr`, this header is blocked.

The `__STDCPP_THREADS__` macro is defined as a nonzero value to indicate that threads are supported by this header.

Members

Public Classes

NAME	DESCRIPTION
thread Class	Defines an object that is used to observe and manage a thread of execution in an application.

Public Structures

NAME	DESCRIPTION
hash Structure (C++ Standard Library)	Defines a member function that returns a value that is uniquely determined by a <code>thread::id</code> . The member function defines a hash function that is suitable for mapping values of type <code>thread::id</code> to a distribution of index values.

Public Functions

NAME	DESCRIPTION
get_id	Uniquely identifies the current thread of execution.
sleep_for	Blocks the calling thread.
sleep_until	Blocks the calling thread at least until the specified time.
swap	Exchanges the states of two thread objects.

NAME	DESCRIPTION
<code>yield</code>	Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

Public Operators

NAME	DESCRIPTION
<code>operator>= Operator</code>	Determines whether one <code>thread::id</code> object is greater than or equal to another.
<code>operator> Operator</code>	Determines whether one <code>thread::id</code> object is greater than another.
<code>operator<= Operator</code>	Determines whether one <code>thread::id</code> object is less than or equal to another.
<code>operator< Operator</code>	Determines whether one <code>thread::id</code> object is less than another.
<code>operator!= Operator</code>	Compares two <code>thread::id</code> objects for inequality.
<code>operator== Operator</code>	Compares two <code>thread::id</code> objects for equality.
<code>operator<< Operator</code>	Inserts a text representation of a <code>thread::id</code> object into a stream.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<thread> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

get_id	sleep_for	sleep_until
swap	yield	

get_id

Uniquely identifies the current thread of execution.

```
thread::id this_thread::get_id() noexcept;
```

Return Value

An object of type [thread::id](#) that uniquely identifies the current thread of execution.

sleep_for

Blocks the calling thread.

```
template <class Rep,  
class Period>  
inline void sleep_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time

A [duration](#) object that specifies a time interval.

Remarks

The function blocks the calling thread for at least the time that's specified by *Rel_time*. This function does not throw any exceptions.

sleep_until

Blocks the calling thread at least until the specified time.

```
template <class Clock, class Duration>  
void sleep_until(const chrono::time_point<Clock, Duration>& Abs_time);  
  
void sleep_until(const xtime *Abs_time);
```

Parameters

Abs_time

Represents a point in time.

Remarks

This function does not throw any exceptions.

swap

Swaps the states of two **thread** objects.

```
void swap(thread& Left, thread& Right) noexcept;
```

Parameters

Left

The left **thread** object.

Right

The right **thread** object.

Remarks

The function calls `Left.swap(Right)` .

yield

Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

```
inline void yield() noexcept;
```

See also

[<thread>](#)

<thread> operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<<</code>	<code>operator<=</code>
<code>operator==</code>		

`operator>=`

Determines whether one `thread::id` object is greater than or equal to another.

```
bool operator>= (  
    thread::id Left,  
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

```
!(Left < Right)
```

Remarks

This function does not throw any exceptions.

`operator>`

Determines whether one `thread::id` object is greater than another.

```
bool operator> (  
    thread::id Left,  
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

```
Right < Left
```

Remarks

This function does not throw any exceptions.

operator<=

Determines whether one `thread::id` object is less than or equal to another.

```
bool operator<= (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

```
!(Right < Left)
```

Remarks

This function does not throw any exceptions.

operator<

Determines whether one `thread::id` object is less than another.

```
bool operator<(
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

true if *Left* precedes *Right* in the total ordering; otherwise, **false**.

Remarks

The operator defines a total ordering on all `thread::id` objects. These objects can be used as keys in associative containers.

This function does not throw any exceptions.

operator!=

Compares two `thread::id` objects for inequality.

```
bool operator!= (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

`!(Left == Right)`

Remarks

This function does not throw any exceptions.

operator==

Compares two `thread::id` objects for equality.

```
bool operator== (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left

The left `thread::id` object.

Right

The right `thread::id` object.

Return Value

true if the two objects represent the same thread of execution or if neither object represents a thread of execution; otherwise, **false**.

Remarks

This function does not throw any exceptions.

operator<<

Inserts a text representation of a `thread::id` object into a stream.

```
template <class Elem, class Tr>
basic_ostream<Elem, Tr>& operator<< (
    basic_ostream<Elem, Tr>& Ostr, thread::id Id);
```

Parameters

Ostr

A [basic_ostream](#) object.

Id

A `thread::id` object.

Return Value

Ostr.

Remarks

This function inserts *Id* into *Ostr*.

If two `thread::id` objects compare equal, the inserted text representations of those objects are the same.

See also

[<thread>](#)

hash Structure (C++ Standard Library)

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a member function that returns a value that's uniquely determined by `val`. The member function defines a `hash` function that's suitable for mapping values of type `thread::id` to a distribution of index values.

Syntax

```
template <>
struct hash<thread::id> :
    public unary_function<thread::id, size_t>
{
    size_t operator()(thread::id Val) const;
};
```

Requirements

Header: `<thread>`

Namespace: `std`

See also

[Header Files Reference](#)

[<thread>](#)

[unary_function Struct](#)

thread Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Defines an object that's used to observe and manage a thread of execution within an application.

Syntax

```
class thread;
```

Remarks

You can use a **thread** object to observe and manage a thread of execution within an application. A thread object that's created by using the default constructor is not associated with any thread of execution. A thread object that's constructed by using a callable object creates a new thread of execution and calls the callable object in that thread. Thread objects can be moved but not copied. Therefore, a thread of execution can be associated with only one thread object.

Every thread of execution has a unique identifier of type `thread::id`. The function `this_thread::get_id` returns the identifier of the calling thread. The member function `thread::get_id` returns the identifier of the thread that's managed by a thread object. For a default-constructed thread object, the `thread::get_id` method returns an object that has a value that's the same for all default-constructed thread objects and different from the value that's returned by `this_thread::get_id` for any thread of execution that could be joined at the time of the call.

Members

Public Classes

NAME	DESCRIPTION
<code>thread::id</code> Class	Uniquely identifies the associated thread.

Public Constructors

NAME	DESCRIPTION
<code>thread</code>	Constructs a thread object.

Public Methods

NAME	DESCRIPTION
<code>detach</code>	Detaches the associated thread from the thread object.
<code>get_id</code>	Returns the unique identifier of the associated thread.
<code>hardware_concurrency</code>	Static. Returns an estimate of the number of hardware thread contexts.
<code>join</code>	Blocks until the associated thread completes.

NAME	DESCRIPTION
<code>joinable</code>	Specifies whether the associated thread is joinable.
<code>native_handle</code>	Returns the implementation-specific type that represents the thread handle.
<code>swap</code>	Swaps the object state with a specified thread object.

Public Operators

NAME	DESCRIPTION
<code>thread::operator=</code>	Associates a thread with the current thread object.

Requirements

Header: <thread>

Namespace: std

thread::detach

Detaches the associated thread. The operating system becomes responsible for releasing thread resources on termination.

```
void detach();
```

Remarks

After a call to `detach`, subsequent calls to `get_id` return `id`.

If the thread that's associated with the calling object is not joinable, the function throws a `system_error` that has an error code of `invalid_argument`.

If the thread that's associated with the calling object is invalid, the function throws a `system_error` that has an error code of `no_such_process`.

thread::get_id

Returns a unique identifier for the associated thread.

```
id get_id() const noexcept;
```

Return Value

A `thread::id` object that uniquely identifies the associated thread, or `thread::id()` if no thread is associated with the object.

thread::hardware_concurrency

Static method that returns an estimate of the number of hardware thread contexts.

```
static unsigned int hardware_concurrency() noexcept;
```

Return Value

An estimate of the number of hardware thread contexts. If the value cannot be computed or is not well defined, this method returns 0.

thread::id Class

Provides a unique identifier for each thread of execution in the process.

```
class thread::id {  
    id() noexcept;  
};
```

Remarks

The default constructor creates an object that does not compare equal to the `thread::id` object for any existing thread.

All default-constructed `thread::id` objects compare equal.

thread::join

Blocks until the thread of execution that's associated with the calling object completes.

```
void join();
```

Remarks

If the call succeeds, subsequent calls to `get_id` for the calling object return a default `thread::id` that does not compare equal to the `thread::id` of any existing thread; if the call does not succeed, the value that's returned by `get_id` is unchanged.

thread::joinable

Specifies whether the associated thread is *joinable*.

```
bool joinable() const noexcept;
```

Return Value

true if the associated thread is *joinable*; otherwise, **false**.

Remarks

A thread object is *joinable* if `get_id() != id()`.

thread::native_handle

Returns the implementation-specific type that represents the thread handle. The thread handle can be used in implementation-specific ways.

```
native_handle_type native_handle();
```


Return Value

`native_handle_type` is defined as a `Win32 HANDLE` that's cast as `void *`.

thread::operator=

Associates the thread of a specified object with the current object.

```
thread& operator=(thread&& Other) noexcept;
```

Parameters

Other

A **thread** object.

Return Value

`*this`

Remarks

The method calls `detach` if the calling object is joinable.

After the association is made, `other` is set to a default-constructed state.

thread::swap

Swaps the object state with that of a specified **thread** object.

```
void swap(thread& Other) noexcept;
```

Parameters

Other

A **thread** object.

thread::thread Constructor

Constructs a **thread** object.

```
thread() noexcept;  
template <class Fn, class... Args>  
explicit thread(Fn&& F, Args&&... A);  
  
thread(thread&& Other) noexcept;
```

Parameters

F

An application-defined function to be executed by the thread.

A

A list of arguments to be passed to *F*.

Other

An existing **thread** object.

Remarks

The first constructor constructs an object that's not associated with a thread of execution. The value that's returned

by a call to `get_id` for the constructed object is `thread::id()` .

The second constructor constructs an object that's associated with a new thread of execution and executes the pseudo-function `INVOKE` that's defined in [<functional>](#) . If not enough resources are available to start a new thread, the function throws a [system_error](#) object that has an error code of `resource_unavailable_try_again` . If the call to *F* terminates with an uncaught exception, [terminate](#) is called.

The third constructor constructs an object that's associated with the thread that's associated with `Other` . `Other` is then set to a default-constructed state.

See also

[Header Files Reference](#)

[<thread>](#)

<tuple>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines a template `tuple` whose instances hold objects of varying types.

Syntax

```
#include <tuple>
```

Classes

CLASS	DESCRIPTION
<code>tuple</code>	Wraps a fixed-length sequence of elements.
<code>tuple_element</code> Class	Wraps the type of a <code>tuple</code> element.
<code>tuple_size</code> Class	Wraps <code>tuple</code> element count.

Operators

OPERATOR	DESCRIPTION
<code>operator==</code>	Comparison of <code>tuple</code> objects, equal
<code>operator!=</code>	Comparison of <code>tuple</code> objects, not equal
<code>operator<</code>	Comparison of <code>tuple</code> objects, less than
<code>operator<=</code>	Comparison of <code>tuple</code> objects, less than or equal
<code>operator></code>	Comparison of <code>tuple</code> objects, greater than
<code>operator>=</code>	Comparison of <code>tuple</code> objects, greater than or equal

Functions

FUNCTION	DESCRIPTION
<code>get</code>	Gets an element from a <code>tuple</code> object.
<code>make_tuple</code>	Makes a <code>tuple</code> from element values.
<code>tie</code>	Makes a <code>tuple</code> from element references.

See also

[<array>](#)

<tuple> functions

10/31/2018 • 3 minutes to read • [Edit Online](#)

get	make_tuple	tie

get

Gets an element from a `tuple` object, by index or (in C++14) by type.

```
// by index:
// get reference to Index element of tuple
template <size_t Index, class... Types>
constexpr tuple_element_t<Index, tuple<Types...>& get(tuple<Types...>& Tuple) noexcept;

// get const reference to Index element of tuple
template <size_t Index, class... Types>
constexpr const tuple_element_t<Index, tuple<Types...>& get(const tuple<Types...>& Tuple) noexcept;

// get rvalue reference to Index element of tuple
template <size_t Index, class... Types>
constexpr tuple_element_t<Index, tuple<Types...>&& get(tuple<Types...>&& Tuple) noexcept;

// (C++14) by type:
// get reference to T element of tuple
template <class T, class... Types>
constexpr T& get(tuple<Types...>& Tuple) noexcept;

// get const reference to T element of tuple
template <class T, class... Types>
constexpr const T& get(const tuple<Types...>& Tuple) noexcept;

// get rvalue reference to T element of tuple
template <class T, class... Types>
constexpr T&& get(tuple<Types...>&& Tuple) noexcept;
```

Parameters

Index

The index of the element to get.

Types

The sequence of types declared in the tuple, in declaration order.

T

The type of the element to get.

Tuple

A `std::tuple` that contains any number of elements.

Remarks

The template functions return a reference to the value at index *Index*, or of type *T* in the `tuple` object.

Calling `get<T>(Tuple)` will produce a compiler error if `Tuple` contains more or less than one element of type *T*.

Example

```
#include <tuple>
#include <iostream>
#include <string>

using namespace std;

int main() {
    tuple<int, double, string> tup(0, 1.42, "Call me Tuple");

    // get elements by index
    cout << " " << get<0>(tup);
    cout << " " << get<1>(tup);
    cout << " " << get<2>(tup) << endl;

    // get elements by type
    cout << " " << get<int>(tup);
    cout << " " << get<double>(tup);
    cout << " " << get<string>(tup) << endl;
}
```

```
0 1.42 Call me Tuple
0 1.42 Call me Tuple
```

make_tuple

Makes a `tuple` from element values.

```
template <class T1, class T2, ..., class TN>
    tuple<V1, V2, ..., VN> make_tuple(const T1& t1, const T2& t2, ..., const TN& tN);
```

Parameters

TN

The type of the Nth function parameter.

tN

The value of the Nth function parameter.

Remarks

The template function returns `tuple<V1, V2, ..., VN>(t1, t2, ..., tN)`, where each type `Vi` is `X&` when the corresponding type `Ti` is `cv reference_wrapper<X>`; otherwise, it is `Ti`.

One advantage of `make_tuple` is that the types of objects that are being stored are determined automatically by the compiler and do not have to be explicitly specified. Don't use explicit template arguments such as

`make_tuple<int, int>(1, 2)` when you use `make_tuple` because it is unnecessarily verbose and adds complex rvalue reference problems that might cause compilation failure.

Example

```

// std_tuple_make_tuple.cpp
// compile by using: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0) << std::endl;

    c0 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0) << std::endl;

    return (0);
}

```

```

0 1 2 3
4 5 6 7

```

tie

Makes a `tuple` from element references.

```

template <class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN&> tie(T1& t1, T2& t2, ..., TN& tN);

```

Parameters

TN

The base type of the Nth tuple element.

Remarks

The template function returns `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tN)`.

Example

```

// std_tuple_tie.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    int v4 = 4;
    double v5 = 5;
    int v6 = 6;
    double v7 = 7;
    std::tie(v4, v5, v6, v7) = c0;

    // display contents " 0 1 2 3"
    std::cout << " " << v4;
    std::cout << " " << v5;
    std::cout << " " << v6;
    std::cout << " " << v7;
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3

```

See also

[<tuple>](#)

<tuple> operators

10/31/2018 • 6 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Compare `tuple` objects for inequality.

```
template <class T1, class T2, ..., class TN,  
          class U1, class U2, ..., class UN>  
bool operator!=(const tuple<T1, T2, ..., TN>& tp1,  
                const tuple<U1, U2, ..., UN>& tp2);
```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns false when `N` is 0, otherwise

```
get<0>(tp1) != get<0>(tp2) || get<1>(tp1) != get<1>(tp2) || ... || get<N - 1>(tp1) == get<N - 1>(tp2) .
```

Example


```

// std_tuple_operator_ne.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 != c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 != c1);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
false
true

```

operator<

Compare `tuple` objects for less.

```

template <class T1, class T2, ..., class TN,
          class U1, class U2, ..., class UN>
bool operator<(const tuple<T1, T2, ..., TN>& tp1,
              const tuple<U1, U2, ..., UN>& tp2);

```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns true when `N` is greater than 0 and the first differing value in `tp1` compares less than the corresponding value in `tp2`, otherwise it returns false.

Example

```

// std_tuple_operator_lt.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 < c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 < c1);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
false
true

```

operator<=

Compare `tuple` objects for less or equal.

```

template <class T1, class T2, ..., class TN,
          class U1, class U2, ..., class UN>
bool operator<=(const tuple<T1, T2, ..., TN>& tp1,
               const tuple<U1, U2, ..., UN>& tp2);

```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns `!(tp2 < tp1)`.

Example

```

// std_tuple_operator_le.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 <= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 <= c0);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
true
false

```

operator==

Compare `tuple` objects for equality.

```

template <class T1, class T2, ..., class TN,
          class U1, class U2, ..., class UN>
bool operator==(const tuple<T1, T2, ..., TN>& tp1,
               const tuple<U1, U2, ..., UN>& tp2);

```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns true when `N` is 0, otherwise

```
get<0>(tp1) == get<0>(tp2) && get<1>(tp1) == get<1>(tp2) && ... && get<N - 1>(tp1) == get<N - 1>(tp2) .
```

Example

```

// std_tuple_operator_eq.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 == c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 == c1);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
true
false

```

operator>

Compare `tuple` objects for greater.

```

template <class T1, class T2, ..., class TN,
          class U1, class U2, ..., class UN>
bool operator>(const tuple<T1, T2, ..., TN>& tp11,
               const tuple<U1, U2, ..., UN>& tp12);

```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns `tp12 < tp11` .

Example

```

// std_tuple_operator_gt.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 > c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 > c0);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
false
true

```

operator>=

Compare `tuple` objects for greater or equal.

```

template <class T1, class T2, ..., class TN,
          class U1, class U2, ..., class UN>
bool operator>=(const tuple<T1, T2, ..., TN>& tp1,
               const tuple<U1, U2, ..., UN>& tp2);

```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns `!(tp1 < tp2)`.

Example

```

// std_tuple_operator_ge.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

    // display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    // display results of comparisons
    std::cout << std::boolalpha << " " << (c0 >= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 >= c1);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
true
false

```

See also

[<tuple>](#)

tuple Class

11/9/2018 • 5 minutes to read • [Edit Online](#)

Wraps a fixed-length sequence of elements.

Syntax

```
class tuple {
public:
    tuple();
    explicit tuple(P1, P2, ..., PN); // 0 < N
    tuple(const tuple&);
    template <class U1, class U2, ..., class UN>
        tuple(const tuple<U1, U2, ..., UN>&);
    template <class U1, class U2>
        tuple(const pair<U1, U2>&); // N == 2

    void swap(tuple& right);
    tuple& operator=(const tuple&);
    template <class U1, class U2, ..., class UN>
        tuple& operator=(const tuple<U1, U2, ..., UN>&);
    template <class U1, class U2>
        tuple& operator=(const pair<U1, U2>&); // N == 2
};
```

Parameters

T_N

The type of the Nth tuple element.

Remarks

The template class describes an object that stores N objects of types *T₁*, *T₂*, ..., *T_N*, respectively, where $0 \leq N \leq N_{\max}$. The extent of a tuple instance `tuple<T1, T2, ..., TN>` is the number *N* of its template arguments. The index of the template argument *T_i* and of the corresponding stored value of that type is *i* - 1. Thus, while we number the types from 1 to N in this documentation, the corresponding index values range from 0 to N - 1.

Example

```

// tuple.cpp
// compile with: /EHsc

#include <vector>
#include <iomanip>
#include <iostream>
#include <tuple>
#include <string>

using namespace std;

typedef tuple<int, double, string> ids;

void print_ids(const ids& i)
{
    cout << "( "
        << get<0>(i) << ", "
        << get<1>(i) << ", "
        << get<2>(i) << " )." << endl;
}

int main( )
{
    // Using the constructor to declare and initialize a tuple
    ids p1(10, 1.1e-2, "one");

    // Compare using the helper function to declare and initialize a tuple
    ids p2;
    p2 = make_tuple(10, 2.22e-1, "two");

    // Making a copy of a tuple
    ids p3(p1);

    cout.precision(3);
    cout << "The tuple p1 is: ( ";
    print_ids(p1);
    cout << "The tuple p2 is: ( ";
    print_ids(p2);
    cout << "The tuple p3 is: ( ";
    print_ids(p3);

    vector<ids> v;

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(make_tuple(3, 3.3e-2, "three"));

    cout << "The tuples in the vector are" << endl;
    for(vector<ids>::const_iterator i = v.begin(); i != v.end(); ++i)
    {
        print_ids(*i);
    }
}

```

```

The tuple p1 is: ( 10, 0.011, one ).
The tuple p2 is: ( 10, 0.222, two ).
The tuple p3 is: ( 10, 0.011, one ).
The tuples in the vector are
( 10, 0.011, one ).
( 10, 0.222, two ).
( 3, 0.033, three ).

```

Requirements

Header: <tuple>

Namespace: std

tuple::operator=

Assigns a `tuple` object.

```
tuple& operator=(const tuple& right);

template <class U1, class U2, ..., class UN>
    tuple& operator=(const tuple<U1, U2, ..., UN>& right);

template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>& right); // N == 2

tuple& operator=(tuple&& right);

template <class U1, class U2>
    tuple& operator=(pair<U1, U2>&& right);
```

Parameters

UN

The type of the Nth copied tuple element.

right

The tuple to copy from.

Remarks

The first two member operators assign the elements of *right* to the corresponding elements of `*this`. The third member operator assigns `right.first` to the element at index 0 of `*this` and `right.second` to the element at index 1. All three member operators return `*this`.

The remaining member operators are analogs to earlier ones, but with [Rvalue Reference Declarator: &&](#).

Example

```

// std_tuple_tuple_operator_as.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>
#include <utility>

typedef std::tuple<int, double, int, double> Mytuple;
int main()
{
    Mytuple c0(0, 1, 2, 3);

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1;
    c1 = c0;

    // display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c1);
    std::cout << " " << std::get<1>(c1);
    std::cout << " " << std::get<2>(c1);
    std::cout << " " << std::get<3>(c1);
    std::cout << std::endl;

    std::tuple<char, int> c2;
    c2 = std::make_pair('x', 4);

    // display contents " x 4"
    std::cout << " " << std::get<0>(c2);
    std::cout << " " << std::get<1>(c2);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
x 4

```

tuple:swap

Exchanges the elements of two tuples.

```

template <class... Types>
void swap(tuple<Types...&> left, tuple<Types...&> right);

```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	A tuple whose elements are to be exchanged with those of the tuple <i>right</i> .
<i>right</i>	A tuple whose elements are to be exchanged with those of the tuple <i>left</i> .

Remarks

The function executes `left.swap(right)` .

tuple::tuple

Constructs a `tuple` object.

```
constexpr tuple();
explicit constexpr tuple(const Types&...);
template <class... UTypes>
    explicit constexpr tuple(UTypes&&...);

tuple(const tuple&) = default;
tuple(tuple&&) = default;

template <class... UTypes>
    constexpr tuple(const tuple<UTypes...>&);
template <class... UTypes>
    constexpr tuple(tuple<UTypes...>&&);

// only if sizeof...(Types) == 2
template <class U1, class U2>
    constexpr tuple(const pair<U1, U2>&);
template <class U1, class U2>
    constexpr tuple(pair<U1, U2>&&);
```

Parameters

UN

The type of the Nth copied tuple element.

right

The tuple to copy from.

Remarks

The first constructor constructs an object whose elements are default constructed.

The second constructor constructs an object whose elements are copy constructed from the arguments `P1` , `P2` , ..., `PN` with each `Pi` initializing the element at index `i - 1` .

The third and fourth constructors construct an object whose elements are copy constructed from the corresponding element of *right*.

The fifth constructor constructs an object whose element at index 0 is copy constructed from `right.first` and whose element at index 1 is copy constructed from `right.second` .

The remaining constructors are analogs to earlier ones, but with [Rvalue Reference Declarator: &&](#).

Example

```

// std_tuple_tuple_tuple.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>
#include <utility>

typedef std::tuple<int, double, int, double> Mytuple;
int main()
{
    Mytuple c0(0, 1, 2, 3);

    // display contents "0 1 2 3"
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1;
    c1 = c0;

    // display contents "0 1 2 3"
    std::cout << std::get<0>(c1) << " ";
    std::cout << std::get<1>(c1) << " ";
    std::cout << std::get<2>(c1) << " ";
    std::cout << std::get<3>(c1);
    std::cout << std::endl;

    std::tuple<char, int> c2(std::make_pair('x', 4));

    // display contents "x 4"
    std::cout << std::get<0>(c2) << " ";
    std::cout << std::get<1>(c2);
    std::cout << std::endl;

    Mytuple c3(c0);

    // display contents "0 1 2 3"
    std::cout << std::get<0>(c3) << " ";
    std::cout << std::get<1>(c3) << " ";
    std::cout << std::get<2>(c3) << " ";
    std::cout << std::get<3>(c3);
    std::cout << std::endl;

    typedef std::tuple<int, float, int, float> Mytuple2;
    Mytuple c4(Mytuple2(4, 5, 6, 7));

    // display contents "4 5 6 7"
    std::cout << std::get<0>(c4) << " ";
    std::cout << std::get<1>(c4) << " ";
    std::cout << std::get<2>(c4) << " ";
    std::cout << std::get<3>(c4);
    std::cout << std::endl;

    return (0);
}

```

```

0 1 2 3
0 1 2 3
x 4
0 1 2 3
4 5 6 7

```

See also

[<tuple>](#)

[make_tuple](#)

tuple_element Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Wraps a `tuple` element. Specializations wrap `array` elements and `pair` elements.

Syntax

```
// CLASS tuple_element (find element by index)
template <size_t Index, class Tuple>
    struct tuple_element;

// tuple_element for const
template <size_t Index, class Tuple>
    struct tuple_element<Index, const Tuple>;

// tuple_element for volatile
template <size_t Index, class Tuple>
    struct tuple_element<Index, volatile Tuple>;

// tuple_element for const volatile
template <size_t Index, class Tuple>
    struct tuple_element<Index, const volatile Tuple>;

// Helper typedef
template <size_t Index, class Tuple>
    using tuple_element_t = typename tuple_element<Index, Tuple>::type;

// Specialization for arrays
template <size_t Index, class Elem, size_t Size>
    struct tuple_element<Index, array<Elem, Size>>;

// Specializations for pairs
// struct to determine type of element 0 in pair
template <class T1, class T2>
    struct tuple_element<0, pair<T1, T2>>;

// struct to determine type of element 1 in pair
template <class T1, class T2>
    struct tuple_element<1, pair<T1, T2>>;
```

Parameters

Index

The index of the designated element.

Tuple

The type of the tuple.

Elem

The type of an array element.

Size

The size of the array.

T1

The type of the first element in a pair.

T2

The type of the second element in a pair.

Remarks

The template class `tuple_element` has a nested typedef `type` that is a synonym for the type at index *Index* of the tuple type *Tuple*.

The typedef `tuple_element_t` is a convenient alias for `tuple_element<Index, Tuple>::type`.

The template class specialization for arrays provides an interface to an `array` as a tuple of `Size` elements, each of which has the same type. Each specialization has a nested typedef `type` that is a synonym for the type of the *Index* element of the `array`, with any const-volatile qualifications preserved.

The template specializations for `pair` types each provide a single member typedef, `type`, which is a synonym for the type of the element at the specified position in the pair, with any const and/or volatile qualifications preserved. The typedef `tuple_element_t` is a convenient alias for `tuple_element<N, pair<T1, T2>>::type`.

Use the [get Function <utility>](#) to return the element at a specified position, or of a specified type.

Example

```
#include <tuple>
#include <string>
#include <iostream>

using namespace std;
typedef tuple<int, double, string> MyTuple;

int main() {
    MyTuple c0{ 0, 1.5, "Tail" };

    tuple_element_t<0, MyTuple> val = get<0>(c0); //get by index
    tuple_element_t<1, MyTuple> val2 = get<1>(c0);
    tuple_element_t<2, MyTuple> val3 = get<string>(c0); // get by type

    cout << val << " " << val2 << " " << val3 << endl;
}
```

```
0 1.5 Tail
```

Example

```

#include <array>
#include <iostream>

using namespace std;
typedef array<int, 4> MyArray;

int main()
{
    MyArray c0 { 0, 1, 2, 3 };

    for (const auto& e : c0)
    {
        cout << e << " ";
    }
    cout << endl;

    // display first element "0"
    tuple_element<0, MyArray>::type val = c0.front();
    cout << val << endl;
}

```

```

0 1 2 3
0

```

Example

```

#include <utility>
#include <iostream>

using namespace std;

typedef pair<int, double> MyPair;
int main() {
    MyPair c0(0, 1.333);

    // display contents "0 1"
    cout << get<0>(c0) << " ";
    cout << get<1>(c0) << endl;

    // display first element "0 " by index
    tuple_element<0, MyPair>::type val = get<0>(c0);
    cout << val << " ";

    // display second element by type
    tuple_element<1, MyPair>::type val2 = get<double>(c0);
    cout << val2 << endl;
}

```

```

0 1.333
0 1.333

```

Requirements

Header: <tuple>

Header: <array> (for array specialization)

Header: <utility> (for pair specializations)

Namespace: std

See also

[tuple](#)

tuple_size Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Reports the number of elements that a `tuple` contains.

Syntax

```
// TEMPLATE STRUCT tuple_size
template <class Tuple>
    struct tuple_size;

// number of elements in array
template <class Elem, size_t Size>
    struct tuple_size<array<Elem, Size>>
        : integral_constant<size_t, Size>;

// size of pair
template <class T1, class T2>
    struct tuple_size<pair<T1, T2>>
        : integral_constant<size_t, 2>;

// size of tuple
template <class... Types>
    struct tuple_size<tuple<Types...>>
        : integral_constant<size_t, sizeof...(Types)>;

// size of const tuple
template <class Tuple>
    struct tuple_size<const Tuple>;

// size of volatile tuple
template <class Tuple>
    struct tuple_size<volatile Tuple>;

// size of const volatile tuple
template <class Tuple>
    struct tuple_size<const volatile Tuple>;
```

Parameters

Tuple

The type of the tuple.

Elem

The type of the array elements.

Size

The size of the array.

T1

The type of the first member of the pair.

T2

The type of the second member of the pair.

Types

The types of the tuple elements.

Remarks

The template class has a member `value` that is an integral constant expression whose value is the extent of the tuple type *Tuple*.

The template specialization for arrays has a member `value` that is an integral constant expression whose value is *Size*, which is the size of the array.

The template specialization for pair has a member `value` that is an integral constant expression whose value is 2.

Example

```
#include <tuple>
#include <iostream>

using namespace std;

typedef tuple<int, double, int, double> MyTuple;
int main()
{
    MyTuple c0(0, 1.5, 2, 3.7);

    // display contents "0 1 2 3"
    cout << get<0>(c0);
    cout << " " << get<1>(c0);
    cout << " " << get<2>(c0);
    cout << " " << get<3>(c0) << endl;

    // display size "4"
    cout << " " << tuple_size<MyTuple>::value << endl;
}
```

```
0 1.5 2 3.7
4
```

Requirements

Header: `<tuple>`

Header: `<array>` (for array specialization)

Header: `<utility>` (for pair specialization)

Namespace: `std`

See also

[<tuple>](#)

[tuple](#)

[tuple_element Class](#)

<type_traits>

2/28/2019 • 6 minutes to read • [Edit Online](#)

Defines templates for compile-time constants that give information about the properties of their type arguments, or produce transformed types.

Syntax

```
#include <type_traits>
```

Remarks

The classes and templates in <type_traits> are used to support type inference, classification, and transformation at compile time. They are also used to detect type-related errors, and to help you optimize your generic code. Unary type traits describe a property of a type, binary type traits describe a relationship between types, and transformation traits modify a property of a type.

The helper class `integral_constant` and its template specializations `true_type` and `false_type` form the base classes for type predicates. A *type predicate* is a template that takes one or more type arguments. When a type predicate *holds true*, it's publicly derived, directly or indirectly, from `true_type`. When a type predicate *holds false*, it's publicly derived, directly or indirectly, from `false_type`.

A *type modifier* or *transformation trait* is a template that takes one or more template arguments and has one member, `type`, which is a synonym for the modified type.

Alias Templates

To simplify type traits expressions, [alias templates](#) for `typename some_trait<T>::type` are provided, where *some_trait* is the template class name. For example, `add_const` has an alias template for its type, `add_const_t`, defined as:

```
template <class T>
using add_const_t = typename add_const<T>::type;
```

These are the provided aliases for the `type` members:

<code>add_const_t</code>	<code>add_cv_t</code>	<code>add_lvalue_reference_t</code>
<code>add_pointer_t</code>	<code>add_rvalue_reference_t</code>	<code>add_volatile_t</code>
<code>aligned_storage_t</code>	<code>aligned_union_t</code>	<code>common_type_t</code>
<code>conditional_t</code>	<code>decay_t</code>	<code>enable_if_t</code>
<code>invoke_result_t</code>	<code>make_signed_t</code>	<code>make_unsigned_t</code>
<code>remove_all_extents_t</code>	<code>remove_const_t</code>	<code>remove_cv_t</code>

remove_extent_t	remove_pointer_t	remove_reference_t
remove_volatile_t	result_of_t	underlying_type_t

Classes

Helper class and typedefs

integral_constant	Makes an integral constant from a type and a value.
true_type	Holds integral constant with true value.
false_type	Holds integral constant with false value.

Primary type categories

is_void	Tests whether the type is void .
is_null_pointer	Tests whether the type is <code>std::nullptr_t</code> .
is_integral	Tests whether the type is integral.
is_floating_point	Tests whether the type is floating-point.
is_array	Tests whether the type is an array.
is_pointer	Tests whether the type is a pointer.
is_lvalue_reference	Tests if type is an lvalue reference.
is_rvalue_reference	Tests if type is an rvalue reference.
is_member_object_pointer	Tests whether the type is a pointer to a member object.
is_member_function_pointer	Tests whether the type is a pointer to a member function.
is_enum	Tests whether the type is an enumeration.
is_union	Tests whether the type is a union.
is_class	Tests whether the type is a class.
is_function	Tests whether the type is a function type.

Composite type categories

is_reference	Tests whether the type is a reference.
is_arithmetic	Tests whether the type is arithmetic.
is_fundamental	Tests whether the type is void or arithmetic.
is_object	Tests whether the type is an object type.
is_scalar	Tests whether the type is scalar.
is_compound	Tests whether the type is not scalar.
is_member_pointer	Tests whether the type is a pointer to a member.

Type properties

is_const	Tests whether the type is const .
is_volatile	Tests whether the type is volatile .
is_trivial	Tests whether the type is trivial.
is_trivially_copyable	Tests whether the type is trivially copyable.
is_standard_layout	Tests if type is a standard layout type.
is_pod	Tests whether the type is a POD.
is_literal_type	Tests whether the type can be a <code>constexpr</code> variable or used in a <code>constexpr</code> function.
is_empty	Tests whether the type is an empty class.
is_polymorphic	Tests whether the type is a polymorphic class.
is_abstract	Tests whether the type is an abstract class.
is_final	Tests whether the type is a class type marked <code>final</code> .
is_signed	Tests whether the type is a signed integer.
is_unsigned	Tests whether the type is an unsigned integer.
is_constructible	Tests whether the type is constructible using the specified argument types.
is_default_constructible	Tests whether the type has a default constructor.
is_copy_constructible	Tests whether the type has a copy constructor.

<code>is_move_constructible</code>	Tests whether the type has a move constructor.
<code>is_assignable</code>	Tests whether the first type can be assigned a value of the second type.
<code>is_copy_assignable</code>	Tests whether a type can be assigned a const reference value of the type.
<code>is_move_assignable</code>	Tests whether a type can be assigned an rvalue reference of the type.
<code>is_destructible</code>	Tests whether the type is destructible.
<code>is_trivially_constructible</code>	Tests whether the type uses no non-trivial operations when constructed using the specified types.
<code>is_trivially_default_constructible</code>	Tests whether the type uses no non-trivial operations when default constructed.
<code>is_trivially_copy_constructible</code>	Tests whether the type uses no non-trivial operations when copy constructed.
<code>is_trivially_move_constructible</code>	Tests whether the type uses no non-trivial operations when move constructed.
<code>is_trivially_assignable</code>	Tests whether the types are assignable and the assignment uses no non-trivial operations.
<code>is_trivially_copy_assignable</code>	Tests whether the type is copy assignable and the assignment uses no non-trivial operations.
<code>is_trivially_move_assignable</code>	Tests whether the type is move assignable and the assignment uses no non-trivial operations.
<code>is_trivially_destructible</code>	Tests whether the type is destructible and the destructor uses no non-trivial operations.
<code>is_nothrow_constructible</code>	Tests whether the type is constructible and is known not to throw when constructed using the specified types.
<code>is_nothrow_default_constructible</code>	Tests whether the type is default constructible and is known not to throw when default constructed.
<code>is_nothrow_copy_constructible</code>	Tests whether the type is copy constructible and the copy constructor is known not to throw.
<code>is_nothrow_move_constructible</code>	Tests whether the type is move constructible and the move constructor is known not to throw.
<code>is_nothrow_assignable</code>	Tests whether the type is assignable using the specified type and the assignment is known not to throw.

is_nothrow_copy_assignable	Tests whether the type is copy assignable and the assignment is known not to throw.
is_nothrow_move_assignable	Tests whether the type is move assignable and the assignment is known not to throw.
is_nothrow_destructible	Tests whether the type is destructible and the destructor is known not to throw.
<code>has_virtual_destructor</code>	Tests whether the type has a virtual destructor.
is_invocable	Tests whether a callable type can be invoked using the specified argument types. Added in C++17.
is_invocable_r	Tests whether a callable type can be invoked using the specified argument types and the result is convertible to the specified type. Added in C++17.
is_nothrow_invocable	Tests whether a callable type can be invoked using the specified argument types and is known not to throw exceptions. Added in C++17.
is_nothrow_invocable_r	Tests whether a callable type can be invoked using the specified argument types and is known not to throw exceptions, and the result is convertible to the specified type. Added in C++17.

Type property queries

alignment_of	Gets the alignment of a type.
rank	Gets the number of array dimensions.
extent	Gets the number of elements in the specified array dimension.

Type relations

is_same	Tests whether two types are the same.
is_base_of	Tests whether one type is a base of another.
is_convertible	Tests whether one type is convertible to another.

Const-volatile modifications

add_const	Produces a const type from type.
add_volatile	Produces a volatile type from type.
add_cv	Produces a const volatile type from type.
remove_const	Produces a non-const type from type.
remove_volatile	Produces a non-volatile type from type.
remove_cv	Produces a non-const non-volatile type from type.

Reference modifications

add_lvalue_reference	Produces a reference to type from type.
add_rvalue_reference	Produces an rvalue reference to type from type
remove_reference	Produces a non-reference type from type.

Sign modifications

make_signed	Produces the type if signed, or the smallest signed type greater than or equal in size to type.
make_unsigned	Produces the type if unsigned, or the smallest unsigned type greater than or equal in size to type.

Array modifications

remove_all_extents	Produces a non-array type from an array type.
remove_extent	Produces the element type from an array type.

Pointer modifications

add_pointer	Produces a pointer to type from type.
remove_pointer	Produces a type from a pointer to type.

Other transformations

aligned_storage	Allocates uninitialized memory for an aligned type.

aligned_union	Allocates uninitialized memory for an aligned union with a non-trivial constructor or destructor.
common_type	Produces the common type of all the types of the parameter pack.
conditional	If the condition is true, produces the first specified type, otherwise the second specified type.
decay	Produces the type as passed by value. Makes non-reference, non-const, or non-volatile type, or makes a pointer to type.
enable_if	If the condition is true, produces the specified type, otherwise no type.
invoke_result	Determines the return type of the callable type that takes the specified argument types. Added in C++17.
result_of	Determines the return type of the callable type that takes the specified argument types. Added in C++14, deprecated in C++17.
underlying_type	Produces the underlying integral type for an enumeration type.

See also

[<functional>](#)

add_const Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes const type from type.

Syntax

```
template <class Ty>
struct add_const;
```

Parameters

Ty

The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *Ty* if *Ty* is a reference, a function, or a const-qualified type, otherwise `const Ty`.

Example

```
// std__type_traits__add_const.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
{
    std::add_const<int>::type *p = (const int *)0;

    p = p; // to quiet "unused" warning
    std::cout << "add_const<int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
add_const<int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_const Class](#)

add_cv Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes **const volatile** type from type.

Syntax

```
template <class T>
struct add_cv;

template <class T>
using add_cv_t = typename add_cv<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of the modified type `add_cv<T>` has a `type` member **typedef** equivalent to *T* modified by both [add_volatile](#) and [add_const](#), unless *T* already has the cv-qualifiers, is a reference, or is a function.

The `add_cv_t<T>` helper type is a shortcut to access the `add_cv<T>` member typedef `type`.

Example

```

// add_cv.cpp
// compile by using: cl /EHsc /W4 add_cv.cpp
#include <type_traits>
#include <iostream>

struct S {
    void f() {
        std::cout << "invoked non-cv-qualified S.f()" << std::endl;
    }
    void f() const {
        std::cout << "invoked const S.f()" << std::endl;
    }
    void f() volatile {
        std::cout << "invoked volatile S.f()" << std::endl;
    }
    void f() const volatile {
        std::cout << "invoked const volatile S.f()" << std::endl;
    }
};

template <class T>
void invoke() {
    T t;
    ((T *)&t)->f();
}

int main()
{
    invoke<S>();
    invoke<std::add_const<S>::type>();
    invoke<std::add_volatile<S>::type>();
    invoke<std::add_cv<S>::type>();
}

```

```

invoked non-cv-qualified S.f()
invoked const S.f()
invoked volatile S.f()
invoked const volatile S.f()

```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_const Class](#)

[remove_volatile Class](#)

add_lvalue_reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes reference to type from type.

Syntax

```
template <class T>
struct add_lvalue_reference;

template <class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if *T* is an lvalue reference, otherwise `T&`.

Example

```
#include <type_traits>
#include <iostream>

using namespace std;
int main()
{
    int val = 0;
    add_lvalue_reference_t<int> p = (int&)val;
    p = p; // to quiet "unused" warning
    cout << "add_lvalue_reference_t<int> == "
          << typeid(p).name() << endl;

    return (0);
}
```

```
add_lvalue_reference_t<int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[remove_reference Class](#)

add_rvalue_reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Creates an rvalue reference type of the template parameter, if it is an object or function type. Otherwise, because of the semantics of reference collapsing, the type is the same as the template parameter.

Syntax

```
template <class T>
struct add_rvalue_reference;

template <class T>
using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;
```

Parameters

T

The type to modify.

Remarks

The `add_rvalue_reference` class has a member named `type`, which is an alias for the type of an rvalue reference to the template parameter *T*. The semantics of reference collapsing imply that, for non-object and non-function types *T*, `T&&` is a *T*. For example, when *T* is an lvalue reference type, `add_rvalue_reference<T>::type` is the lvalue reference type, not an rvalue reference.

For convenience, `<type_traits>` defines a helper template, `add_rvalue_reference_t`, that aliases the `type` member of `add_rvalue_reference`.

Example

This code example uses `static_assert` to show how rvalue reference types are created by using `add_rvalue_reference` and `add_rvalue_reference_t`, and how the result of `add_rvalue_reference` on an lvalue reference type is not an rvalue reference, but collapses to the lvalue reference type.

```

// ex_add_rvalue_reference.cpp
// Build by using: cl /EHsc /W4 ex_add_rvalue_reference.cpp
#include <type_traits>
#include <iostream>
#include <string>

using namespace std;
int main()
{
    static_assert(is_same<add_rvalue_reference<string>::type, string&&>::value,
        "Expected add_rvalue_reference_t<string> to be string&&");
    static_assert(is_same<add_rvalue_reference_t<string*>, string*&&>::value,
        "Expected add_rvalue_reference_t<string*> to be string*&&");
    static_assert(is_same<add_rvalue_reference<string&>::type, string&>::value,
        "Expected add_rvalue_reference_t<string&> to be string&");
    static_assert(is_same<add_rvalue_reference_t<string&&>, string&&>::value,
        "Expected add_rvalue_reference_t<string&&> to be string&&");
    cout << "All static_assert tests of add_rvalue_reference passed." << endl;
    return 0;
}

/*Output:
All static_assert tests of add_rvalue_reference passed.
*/

```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[add_lvalue_reference Class](#)

[is_rvalue_reference Class](#)

add_pointer Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes a pointer-to-type from a specified type.

Syntax

```
template <class T>
struct add_pointer;

template <class T>
using add_pointer_t = typename add_pointer<T>::type;
```

Parameters

T

The type to modify.

Remarks

The member **typedef** `type` names the same type as `remove_reference<T>::type*`. The alias `add_pointer_t` is a shortcut to access the member **typedef** `type`.

Because it is invalid to make a pointer from a reference, `add_pointer` removes the reference, if any, from the specified type before it makes a pointer-to-type. Consequently, you can use a type with `add_pointer` without being concerned about whether the type is a reference.

Example

The following example demonstrates that `add_pointer` of a type is the same as a pointer to that type.

```
#include <type_traits>
#include <iostream>

int main()
{
    std::add_pointer_t<int> *p = (int **)0;

    p = p; // to quiet "unused" warning
    std::cout << "add_pointer_t<int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
add_pointer_t<int> == int *
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_pointer](#) Class

add_volatile Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes a **volatile** type from the specified type.

Syntax

```
template <class Ty>
struct add_volatile;

template <class T>
using add_volatile_t = typename add_volatile<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `add_volatile<T>` has a member **typedef** `type` that is *T* if *T* is a reference, a function, or a volatile-qualified type, otherwise **volatile** *T*. The alias `add_volatile_t` is a shortcut to access the member **typedef** `type`.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    std::add_volatile_t<int> *p = (volatile int *)0;

    p = p; // to quiet "unused" warning
    std::cout << "add_volatile<int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
add_volatile<int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_volatile Class](#)

aligned_storage Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Makes suitably aligned type.

Syntax

```
template <std::size_t Len, std::size_t Align>
struct aligned_storage;

template <std::size_t Len, std::size_t Align = alignment_of<max_align_t>::value>
using aligned_storage_t = typename aligned_storage<Len, Align>::type;
```

Parameters

Len

The object size.

Align

The object alignment.

Remarks

The template member typedef `type` is a synonym for a POD type with alignment *Align* and size *Len*. *Align* must be equal to `alignment_of<T>::value` for some type `T`, or to the default alignment.

Example

```
#include <type_traits>
#include <iostream>

typedef std::aligned_storage<sizeof (int),
    std::alignment_of<double>::value>::type New_type;
int main()
{
    std::cout << "alignment_of<int> == "
        << std::alignment_of<int>::value << std::endl;
    std::cout << "aligned to double == "
        << std::alignment_of<New_type>::value << std::endl;

    return (0);
}
```

```
alignment_of<int> == 4
aligned to double == 8
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[alignment_of Class](#)

aligned_union Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Provides a POD type large enough and suitably aligned to store a union type, and the size required.

Syntax

```
template <std::size_t Len, class... Types>
struct aligned_union;

template <std::size_t Len, class... Types>
using aligned_union_t = typename aligned_union<Len, Types...>::type;
```

Parameters

Len

The alignment value for the largest type in the union.

Types

The distinct types in the underlying union.

Remarks

Use the template class to get the alignment and size needed to store a union in uninitialized storage. The member typedef `type` names a POD type suitable for storage of any type listed in *Types*; the minimum size is *Len*. The static member `alignment_value` of type `std::size_t` contains the strictest alignment required of all the types listed in *Types*.

Example

The following example shows how to use `aligned_union` to allocate an aligned stack buffer to place a union.

```

// std_type_traits__aligned_union.cpp
#include <iostream>
#include <type_traits>

union U_type
{
    int i;
    float f;
    double d;
    U_type(float e) : f(e) {}
};

typedef std::aligned_union<16, int, float, double>::type aligned_U_type;

int main()
{
    // allocate memory for a U_type aligned on a 16-byte boundary
    aligned_U_type au;
    // do placement new in the aligned memory on the stack
    U_type* u = new (&au) U_type(1.0f);
    if (nullptr != u)
    {
        std::cout << "value of u->i is " << u->i << std::endl;
        // must clean up placement objects manually!
        u->~U_type();
    }
}

```

```

value of u->i is 1065353216

```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[alignment_of Class](#)

alignment_of Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Gets alignment of the specified type. This struct is implemented in terms of [alignof](#). Use `alignof` directly when you simply need to query an alignment value. Use `alignment_of` when you need an integral constant, for example when doing tag dispatch.

Syntax

```
template <class Ty>
struct alignment_of;
```

Parameters

Ty

The type to query.

Remarks

The type query holds the value of the alignment of the type *Ty*.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[aligned_storage](#) Class

common_type Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Determines the common type of one or more types.

Syntax

```
template <class... T>
struct common_type;

template <class T>
struct common_type<T> {
    typedef typename decay<T>::type type;
};

template <class T, class U>
struct common_type<T, U> {
    typedef typename decay<decltype(true declval<T>() :
        declval<U>())>::type type;
};

template <class T, class U, class... V>
struct common_type<T, U, V...> {
    typedef typename common_type<typename common_type<T, U>::type, V...>::type type;
};
```

Parameters

List of types that are either [complete types](#) or void.

Remarks

The `type` member is the common type to which all types in the parameter list can be converted.

Example

The following program demonstrates some correct usage scenarios and tests for results.

```

// Compile using cl.exe /EHsc
// common_type sample
#include <iostream>
#include <type_traits>

struct Base {};
struct Derived : Base {};

int main()
{
    typedef std::common_type<unsigned char, short, int>::type NumericType;
    typedef std::common_type<float, double>::type FloatType;
    typedef std::common_type<const int, volatile int>::type ModifiedIntType;
    typedef std::common_type<Base, Derived>::type ClassType;

    std::cout << std::boolalpha;
    std::cout << "Test for typedefs of common_type int" << std::endl;
    std::cout << "NumericType: " << std::is_same<int, NumericType>::value << std::endl;
    std::cout << "FloatType: " << std::is_same<int, FloatType>::value << std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<int, ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: " << std::is_same<int, ClassType>::value << std::endl;
    std::cout << "-----" << std::endl;
    std::cout << "Test for typedefs of common_type double" << std::endl;
    std::cout << "NumericType: " << std::is_same<double, NumericType>::value << std::endl;
    std::cout << "FloatType: " << std::is_same<double, FloatType>::value << std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<double, ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: " << std::is_same<double, ClassType>::value << std::endl;
    std::cout << "-----" << std::endl;
    std::cout << "Test for typedefs of common_type Base" << std::endl;
    std::cout << "NumericType: " << std::is_same<Base, NumericType>::value << std::endl;
    std::cout << "FloatType: " << std::is_same<Base, FloatType>::value << std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<Base, ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: " << std::is_same<Base, ClassType>::value << std::endl;

    return 0;
}

```

Output

```

Test for typedefs of common_type int
NumericType: true
FloatType: false
ModifiedIntType: true
ClassType: false
-----
Test for typedefs of common_type double
NumericType: false
FloatType: true
ModifiedIntType: false
ClassType: false
-----
Test for typedefs of common_type Base
NumericType: false
FloatType: false
ModifiedIntType: false
ClassType: true

```

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

conditional Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Selects one of two types, depending on the specified condition.

Syntax

```
template <bool B, class T1, class T2>
struct conditional;

template <bool _Test, class _T1, class _T2>
using conditional_t = typename conditional<_Test, _T1, _T2>::type;
```

Parameters

B

The value that determines the selected type.

T1

The type result when *B* is true.

T2

The type result when *B* is false.

Remarks

The template member typedef `conditional<B, T1, T2>::type` evaluates to *T1* when *B* evaluates to **true**, and evaluates to *T2* when *B* evaluates to **false**.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

decay Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Produces the type as passed by value. Makes the type non-reference, non-const, non-volatile, or makes a pointer to the type from a function or an array type.

Syntax

```
template <class T>
struct decay;

template <class T>
using decay_t = typename decay<T>::type;
```

Parameters

T

The type to modify.

Remarks

Use the decay template to produce the resulting type as if the type was passed by value as an argument. The template class member typedef `type` holds a modified type that is defined in the following stages:

- The type `U` is defined as `remove_reference<T>::type`.
- If `is_array<U>::value` is true, the modified type `type` is `remove_extent<U>::type *`.
- Otherwise, if `is_function<U>::value` is true, the modified type `type` is `add_pointer<U>::type`.
- Otherwise, the modified type `type` is `remove_cv<U>::type`.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

enable_if Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

Conditionally makes an instance of a type for SFINAE overload resolution. The nested typedef

`enable_if<Condition,Type>::type` exists—and is a synonym for `Type`—if and only if `Condition` is **true**.

Syntax

```
template <bool B, class T = void>
struct enable_if;
```

Parameters

B

The value that determines the existence of the resulting type.

T

The type to instantiate if *B* is true.

Remarks

If *B* is true, `enable_if<B, T>` has a nested typedef named "type" that's a synonym for *T*.

If *B* is false, `enable_if<B, T>` doesn't have a nested typedef named "type".

This alias template is provided:

```
template <bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

In C++, substitution failure of template parameters is not an error in itself—this is referred to as *SFINAE* (substitution failure is not an error). Typically, `enable_if` is used to remove candidates from overload resolution—that is, it culls the overload set—so that one definition can be rejected in favor of another. This conforms to SFINAE behavior. For more information about SFINAE, see [Substitution failure is not an error](#) on Wikipedia.

Here are four example scenarios:

- Scenario 1: Wrapping the return type of a function:

```
template <your_stuff>
typename enable_if<your_condition, your_return_type>::type
yourfunction(args) { // ...
}
// The alias template makes it more concise:
template <your_stuff>
enable_if_t<your_condition, your_return_type>
yourfunction(args) { // ...
}
```

- Scenario 2: Adding a function parameter that has a default argument:

```
template <your_stuff>
your_return_type_if_present
    yourfunction(args, enable_if_t<your condition, FOO> = BAR) { // ...
}
```

- Scenario 3: Adding a template parameter that has a default argument:

```
template <your_stuff, typename Dummy = enable_if_t<your_condition>>
rest_of_function_declaration_goes_here
```

- Scenario 4: If your function has a non-templated argument, you can wrap its type:

```
template <typename T>
void your_function(const T& t,
    enable_if_t<is_something<T>::value, const string&>
s) { // ...
}
```

Scenario 1 doesn't work with constructors and conversion operators because they don't have return types.

Scenario 2 leaves the parameter unnamed. You could say `::type Dummy = BAR`, but the name `Dummy` is irrelevant, and giving it a name is likely to trigger an "unreferenced parameter" warning. You have to choose a `FOO` function parameter type and `BAR` default argument. You could say `int` and `0`, but then users of your code could accidentally pass to the function an extra integer that would be ignored. Instead, we recommend that you use `void **` and either `0` or `nullptr` because almost nothing is convertible to `void **`:

```
template <your_stuff>
your_return_type_if_present
yourfunction(args, typename enable_if<your_condition, void **>::type = nullptr) { // ...
}
```

Scenario 2 also works for ordinary constructors. However, it doesn't work for conversion operators because they can't take extra parameters. It also doesn't work for [variadic](#) constructors because adding extra parameters makes the function parameter pack a non-deduced context and thereby defeats the purpose of `enable_if`.

Scenario 3 uses the name `Dummy`, but it's optional. Just "`typename = typename`" would work, but if you think that looks weird, you can use a "dummy" name—just don't use one that might also be used in the function definition. If you don't give a type to `enable_if`, it defaults to `void`, and that's perfectly reasonable because you don't care what `Dummy` is. This works for everything, including conversion operators and [variadic](#) constructors.

Scenario 4 works in constructors that don't have return types, and thereby solves the wrapping limitation of Scenario 1. However, Scenario 4 is limited to non-templated function arguments, which aren't always available. (Using Scenario 4 on a templated function argument prevents template argument deduction from working on it.)

`enable_if` is powerful, but also dangerous if it's misused. Because its purpose is to make candidates vanish before overload resolution, when it's misused, its effects can be very confusing. Here are some recommendations:

- Do not use `enable_if` to select between implementations at compile-time. Don't ever write one `enable_if` for `CONDITION` and another for `!CONDITION`. Instead, use a *tag dispatch* pattern—for example, an algorithm that selects implementations depending on the strengths of the iterators they're given.
- Do not use `enable_if` to enforce requirements. If you want to validate template parameters, and if the validation fails, cause an error instead of selecting another implementation, use [static_assert](#).
- Use `enable_if` when you have an overload set that makes otherwise good code ambiguous. Most often,

this occurs in implicitly converting constructors.

Example

This example explains how the C++ Standard Library template function `std::make_pair()` takes advantage of `enable_if`.

```
void func(const pair<int, int>&);

void func(const pair<string, string>&);

func(make_pair("foo", "bar"));
```

In this example, `make_pair("foo", "bar")` returns `pair<const char *, const char *>`. Overload resolution has to determine which `func()` you want. `pair<A, B>` has an implicitly converting constructor from `pair<X, Y>`. This isn't new—it was in C++98. However, in C++98/03, the implicitly converting constructor's signature always exists, even if it's `pair<int, int>(const pair<const char *, const char *>&)`. Overload resolution doesn't care that an attempt to instantiate that constructor explodes horribly because `const char *` isn't implicitly convertible to `int`; it's only looking at signatures, before function definitions are instantiated. Therefore, the example code is ambiguous, because signatures exist to convert `pair<const char *, const char *>` to both `pair<int, int>` and `pair<string, string>`.

C++11 solved this ambiguity by using `enable_if` to make sure `pair<A, B>(const pair<X, Y>&)` exists **only** when `const X&` is implicitly convertible to `A` and `const Y&` is implicitly convertible to `B`. This allows overload resolution to determine that `pair<const char *, const char *>` is not convertible to `pair<int, int>` and that the overload that takes `pair<string, string>` is viable.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[`<type_traits>`](#)

extent Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Gets an array dimension.

Syntax

```
template <class Ty, unsigned I = 0>
struct extent;
```

Parameters

Ty

The type to query.

I

The array bound to query.

Remarks

If *Ty* is an array type that has at least *I* dimensions, the type query holds the number of elements in the dimension specified by *I*. If *Ty* is not an array type or its rank is less than *I*, or if *I* is zero and *Ty* is of type "array of unknown bound of `u`", the type query holds the value 0.

Example

```
// std__type_traits__extent.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << "extent 0 == "
        << std::extent<int[5][10]>::value << std::endl;
    std::cout << "extent 1 == "
        << std::extent<int[5][10], 1>::value << std::endl;

    return (0);
}
```

```
extent 0 == 5
extent 1 == 10
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

<type_traits>

remove_all_extents Class

remove_extent Class

integer_sequence Class

11/8/2018 • 2 minutes to read • [Edit Online](#)

Represents an integer sequence. Can be used to deduce and expand parameter packs in variadic types such as `std::tuple<T...>` that are passed as arguments to a function.

Syntax

```
template <class T, T... Vals>
struct integer_sequence
```

Parameters

T

The type of the values; must be an integral type: `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`, or signed or unsigned integer types.

Vals

A non-type parameter pack that represents a sequence of values of integral type `T`.

Members

<code>static size_t size() noexcept</code>	The number of elements in the sequence.
<code>typedef T value_type</code>	The type of each element in the sequence. Must be an integral type.

Remarks

A parameter pack that is passed directly to a function can be unpacked without any special library helpers. When a parameter pack is part of a type that is passed to a function, and you need indices to access the elements, then the easiest way to unpack it is to use `integer_sequence` and its related type aliases `make_integer_sequence`, `index_sequence`, `make_index_sequence`, and `index_sequence_for`.

Example

The following example is based on the original proposal [N3658](#). It shows how to use an `integer_sequence` to create a `std::tuple` from a `std::array<T,N>`, and how to use an `integer_sequence` to get at the tuple members.

In the `a2t` function, an `index_sequence` is an alias of `integer_sequence` based on the `size_t` integral type. `make_index_sequence` is an alias that at compile time creates a zero-based `index_sequence` with the same number of elements as the array that is passed in by the caller. `a2t` passes the `index_sequence` by value to `a2t_`, where the expression `a[I]...` unpacks `I`, and then the elements are being fed to `make_tuple` which consumes them as individual arguments. For example, if the sequence contains three elements, then `make_tuple` is called as `make_tuple(a[0], a[1], a[2])`. The array elements themselves can of course be any type.

The `apply` function accepts a `std::tuple`, and produces an `integer_sequence` by using the `tuple_size` helper class. Note that `std::decay_t` is necessary because `tuple_size` does not work with reference types. The `apply_` function

unpacks the tuple members and forwards them as separate arguments to a function call. In this example the function is a simple lambda expression that prints out the values.

```
#include <stddef.h>
#include <iostream>
#include <tuple>
#include <utility>
#include <array>
#include <string>

using namespace std;

// Create a tuple from the array and the index_sequence
template<typename Array, size_t... I>
auto a2t_(const Array& a, index_sequence<I...>)
{
    return make_tuple(a[I]...);
}

// Create an index sequence for the array, and pass it to the
// implementation function a2t_
template<typename T, size_t N>
auto a2t(const array<T, N>& a)
{
    return a2t_(a, make_index_sequence<N>());
}

// Call function F with the tuple members as separate arguments.
template<typename F, typename Tuple = tuple<T...>, size_t... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return forward<F>(f)(get<I>(forward<Tuple>(args))...);
}

// Create an index_sequence for the tuple, and pass it with the
// function object and the tuple to the implementation function apply_
template<typename F, typename Tuple = tuple<T...>>
decltype(auto) apply(F&& f, Tuple&& args)
{
    using Indices = make_index_sequence<tuple_size<decay_t<Tuple>>::value >;
    return apply_(forward<F>(f), forward<Tuple>(args), Indices());
}

int main()
{
    const array<string, 3> arr { "Hello", "from", "C++14" };

    //Create a tuple given a array
    auto tup = a2t(arr);

    // Extract the tuple elements
    apply([](const string& a, const string& b, const string& c) {cout << a << " " << b << " " << c << endl; },
tup);

    char c;
    cin >> c;
}
```

To make an `index_sequence` for a parameter pack, use `index_sequence_for<T...>` which is an alias for `make_index_sequence<sizeof...(T)>`

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[Ellipses and Variadic Templates](#)

integral_constant Class, bool_constant Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Makes an integral constant from a type and value.

Syntax

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant<T, v> type;
    constexpr operator value_type() const noexcept;
    constexpr value_type operator()() const noexcept;
};
```

Parameters

T

The type of the constant.

v

The value of the constant.

Remarks

The `integral_constant` template class, when specialized with an integral type *T* and a value *v* of that type, represents an object that holds a constant of that integral type with the specified value. The member named `type` is an alias for the generated template specialization type, and the `value` member holds the value *v* used to create the specialization.

The `bool_constant` template class is an explicit partial specialization of `integral_constant` that uses **bool** as the *T* argument.

Example

```
// std__type_traits__integral_constant.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << "integral_constant<int, 5> == "
               << std::integral_constant<int, 5>::value << std::endl;
    std::cout << "integral_constant<bool, false> == " << std::boolalpha
               << std::integral_constant<bool, false>::value << std::endl;

    return (0);
}
```

```
integral_constant<int, 5> == 5  
integral_constant<bool, false> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[false_type](#)

[true_type](#)

invoke_result Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

Determines the return type of the callable type that takes the specified argument types at compile time. Added in C++17.

Syntax

```
template <class Callable, class... Args>
    struct invoke_result<Callable(Args...)>;

// Helper type
template<class Callable, class... Args>
    using invoke_result_t = typename invoke_result<Callable, Args...>::type;
```

Parameters

Callable

The callable type to query.

Args

The types of the argument list to the callable type to query.

Remarks

Use this template to determine the result type of *Callable(Args...)* at compile time, where *Callable* and all types in *Args* are any complete type, an array of unknown bound, or a possibly cv-qualified `void`. The `type` member of the template class names the return type of *Callable* when invoked using the arguments *Args...*. The `type` member is only defined if *Callable* can be called when invoked using the arguments *Args...* in an unevaluated context. Otherwise, the template class has no member `type`, which allows SFINAE tests on a particular set of argument types at compile time.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[invoke](#)

is_abstract Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is abstract class.

Syntax

```
template <class Ty>
struct is_abstract;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has at least one pure virtual function, otherwise it holds false.

Example

```
// std__type_traits__is_abstract.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct abstract
{
    virtual int val() = 0;
};

int main()
{
    std::cout << "is_abstract<trivial> == " << std::boolalpha
        << std::is_abstract<trivial>::value << std::endl;
    std::cout << "is_abstract<abstract> == " << std::boolalpha
        << std::is_abstract<abstract>::value << std::endl;

    return (0);
}
```

```
is_abstract<trivial> == false
is_abstract<abstract> == true
```

Requirements

Header: <type_traits>

Namespace: `std`

See also

[<type_traits>](#)

[is_polymorphic](#) Class

is_arithmetic Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is arithmetic.

Syntax

```
template <class Ty>
struct is_arithmetic;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an arithmetic type, that is, an integral type or a floating point type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

```
// std__type_traits__is_arithmetic.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_arithmetic<trivial> == " << std::boolalpha
        << std::is_arithmetic<trivial>::value << std::endl;
    std::cout << "is_arithmetic<int> == " << std::boolalpha
        << std::is_arithmetic<int>::value << std::endl;
    std::cout << "is_arithmetic<float> == " << std::boolalpha
        << std::is_arithmetic<float>::value << std::endl;

    return (0);
}
```

```
is_arithmetic<trivial> == false
is_arithmetic<int> == true
is_arithmetic<float> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_floating_point](#) Class

[is_integral](#) Class

is_array Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is array.

Syntax

```
template <class Ty>
struct is_array;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an array type, otherwise it holds false.

Example

```
// std__type_traits__is_array.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_array<trivial> == " << std::boolalpha
        << std::is_array<trivial>::value << std::endl;
    std::cout << "is_array<int> == " << std::boolalpha
        << std::is_array<int>::value << std::endl;
    std::cout << "is_array<int[5]> == " << std::boolalpha
        << std::is_array<int[5]>::value << std::endl;

    return (0);
}
```

```
is_array<trivial> == false
is_array<int> == false
is_array<int[5]> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[extent Class](#)

[rank Class](#)

is_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a value of `From` type can be assigned to a `To` type.

Syntax

```
template <class To, class From>
struct is_assignable;
```

Parameters

To

The type of the object that receives the assignment.

From

The type of the object that provides the value.

Remarks

The unevaluated expression `declval<To>() = declval<From>()` must be well-formed. Both `From` and `To` must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_base_of Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether one type is base of another.

Syntax

```
template <class Base, class Derived>
struct is_base_of;
```

Parameters

Base

The base class to test for.

Derived

The derived type to test for.

Remarks

An instance of the type predicate holds true if the type *Base* is a base class of the type *Derived*, otherwise it holds false.

Example

```
#include <type_traits>
#include <iostream>

struct base
{
    int val;
};

struct derived
    : public base
{
};

int main()
{
    std::cout << "is_base_of<base, base> == " << std::boolalpha
        << std::is_base_of<base, base>::value << std::endl;
    std::cout << "is_base_of<base, derived> == " << std::boolalpha
        << std::is_base_of<base, derived>::value << std::endl;
    std::cout << "is_base_of<derived, base> == " << std::boolalpha
        << std::is_base_of<derived, base>::value << std::endl;

    return (0);
}
```

```
is_base_of<base, base> == true
is_base_of<base, derived> == true
is_base_of<derived, base> == false
```


Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_convertible](#) Class

is_class Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a class.

Syntax

```
template <class Ty>
struct is_class;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a type defined as a **class** or a **struct**, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

```
// std__type_traits__is_class.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_class<trivial> == " << std::boolalpha
        << std::is_class<trivial>::value << std::endl;
    std::cout << "is_class<int> == " << std::boolalpha
        << std::is_class<int>::value << std::endl;

    return (0);
}
```

```
is_class<trivial> == true
is_class<int> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

<type_traits>

is_compound Class

is_union Class

is_compound Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if the specified type is not fundamental.

Syntax

```
template <class Ty>
struct is_compound;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds **false** if the type of *Ty* is a fundamental type (that is, if [is_fundamental](#)<*Ty*> holds **true**); otherwise, it holds **true**. Thus, the predicate holds **true** if *Ty* is an array type, a function type, a pointer to **void** or an object or a function, a reference, a class, a union, an enumeration, or a pointer to non-static class member, or a *cv-qualified* form of one of them.

Example

```
// std__type_traits__is_compound.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_compound<trivial> == " << std::boolalpha
        << std::is_compound<trivial>::value << std::endl;
    std::cout << "is_compound<int[]> == " << std::boolalpha
        << std::is_compound<int[]>::value << std::endl;
    std::cout << "is_compound<int()> == " << std::boolalpha
        << std::is_compound<int()>::value << std::endl;
    std::cout << "is_compound<int*> == " << std::boolalpha
        << std::is_compound<int*>::value << std::endl;
    std::cout << "is_compound<void *> == " << std::boolalpha
        << std::is_compound<void *>::value << std::endl;
    std::cout << "is_compound<int> == " << std::boolalpha
        << std::is_compound<int>::value << std::endl;

    return (0);
}
```

```
is_compound<trivial> == true
is_compound<int[]> == true
is_compound<int()> == true
is_compound<int&> == true
is_compound<void *> == true
is_compound<int> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_class](#) [Class](#)

is_const Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is const.

Syntax

```
template <class Ty>
struct is_const;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if *Ty* is `const-qualified`.

Example

```
// std__type_traits__is_const.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_const<trivial> == " << std::boolalpha
        << std::is_const<trivial>::value << std::endl;
    std::cout << "is_const<const trivial> == " << std::boolalpha
        << std::is_const<const trivial>::value << std::endl;
    std::cout << "is_const<int> == " << std::boolalpha
        << std::is_const<int>::value << std::endl;
    std::cout << "is_const<const int> == " << std::boolalpha
        << std::is_const<const int>::value << std::endl;

    return (0);
}
```

```
is_const<trivial> == false
is_const<const trivial> == true
is_const<int> == false
is_const<const int> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_volatile Class](#)

is_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a type is constructible when the specified argument types are used.

Syntax

```
template <class T, class... Args>
struct is_constructible;
```

Parameters

T

The type to query.

Args

The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is constructible by using the argument types in *Args*, otherwise it holds false. Type *T* is constructible if the variable definition `T t(std::declval<Args>()...);` is well-formed. Both *T* and all the types in *Args* must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_convertible Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if one type is convertible to another.

Syntax

```
template <class From, class To>
struct is_convertible;
```

Parameters

From

The type to convert from.

To

The type to convert to.

Remarks

An instance of the type predicate holds true if the expression `To to = from;`, where `from` is an object of type `From`, is well-formed.

Example

```
// std_type_traits_is_convertible.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_convertible<trivial, int> == " << std::boolalpha
        << std::is_convertible<trivial, int>::value << std::endl;
    std::cout << "is_convertible<trivial, trivial> == " << std::boolalpha
        << std::is_convertible<trivial, trivial>::value << std::endl;
    std::cout << "is_convertible<char, int> == " << std::boolalpha
        << std::is_convertible<char, int>::value << std::endl;

    return (0);
}
```

```
is_convertible<trivial, int> == false
is_convertible<trivial, trivial> == true
is_convertible<char, int> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_base_of](#) `Class`

is_copy_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has can be copied on assignment.

Syntax

```
template <class Ty>
struct is_copy_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy assignment operator, otherwise it holds false. Equivalent to `is_assignable<Ty&, const Ty&>`.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_copy_constructible Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type has a copy constructor.

Syntax

```
template <class Ty>
struct is_copy_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy constructor, otherwise it holds false.

Example

```
#include <type_traits>
#include <iostream>

struct Copyable
{
    int val;
};

struct NotCopyable
{
    NotCopyable(const NotCopyable&) = delete;
    int val;
};

int main()
{
    std::cout << "is_copy_constructible<Copyable> == " << std::boolalpha
              << std::is_copy_constructible<Copyable>::value << std::endl;
    std::cout << "is_copy_constructible<NotCopyable> == " << std::boolalpha
              << std::is_copy_constructible<NotCopyable>::value << std::endl;

    return (0);
}
```

```
is_copy_constructible<Copyable> == true
is_copy_constructible<NotCopyable> == false
```

Requirements

Header: <type_traits>

Namespace: `std`

See also

[`<type_traits>`](#)

is_default_constructible Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if a type has a default constructor.

Syntax

```
template <class Ty>
struct is_default_constructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type that has a default constructor, otherwise it holds false. This is equivalent to the predicate `is_constructible<T>`. Type *T* must be a complete type, **void**, or an array of unknown bound.

Example

```
#include <type_traits>
#include <iostream>

struct Simple
{
    Simple() : val(0) {}
    int val;
};

struct Simple2
{
    Simple2(int v) : val(v) {}
    int val;
};

int main()
{
    std::cout << "is_default_constructible<Simple> == " << std::boolalpha
        << std::is_default_constructible<Simple>::value << std::endl;
    std::cout << "is_default_constructible<Simple2> == " << std::boolalpha
        << std::is_default_constructible<Simple2>::value << std::endl;

    return (0);
}
```

```
is_default_constructible<Simple> == true
is_default_constructible<Simple2> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_destructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is destructible.

Syntax

```
template <class T>
struct is_destructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, otherwise it holds false. Destructible types are reference types, object types, and types where for some type `U` equal to `remove_all_extents_t<T>` the unevaluated operand `std::declval<U&>::~~U()` is well-formed. Other types, including incomplete types, **void**, and function types, are not destructible types.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_empty Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is an empty class.

Syntax

```
template <class Ty>
struct is_empty;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an empty class, otherwise it holds false.

Example

```
// std__type_traits__is_empty.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct empty
{
};

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_empty<trivial> == " << std::boolalpha
        << std::is_empty<trivial>::value << std::endl;
    std::cout << "is_empty<empty> == " << std::boolalpha
        << std::is_empty<empty>::value << std::endl;
    std::cout << "is_empty<int> == " << std::boolalpha
        << std::is_empty<int>::value << std::endl;

    return (0);
}
```

```
is_empty<trivial> == false
is_empty<empty> == true
is_empty<int> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_enum Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is an enumeration.

Syntax

```
template <class Ty>
struct is_enum;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an enumeration type or a `cv-qualified` form of an enumeration type, otherwise it holds false.

Example

```
// std__type_traits__is_enum.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

enum color {
    red, greed, blue};

int main()
{
    std::cout << "is_enum<trivial> == " << std::boolalpha
              << std::is_enum<trivial>::value << std::endl;
    std::cout << "is_enum<color> == " << std::boolalpha
              << std::is_enum<color>::value << std::endl;
    std::cout << "is_enum<int> == " << std::boolalpha
              << std::is_enum<int>::value << std::endl;

    return (0);
}
```

```
is_enum<trivial> == false
is_enum<color> == true
is_enum<int> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_integral](#) Class

is_final Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is a class type marked `final`.

Syntax

```
template <class T>
struct is_final;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type marked `final`, otherwise it holds false. If *T* is a class type, it must be a complete type.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[final Specifier](#)

is_floating_point Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is floating point.

Syntax

```
template <class Ty>
struct is_floating_point;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a floating point type or a `cv-qualified` form of a floating point type, otherwise it holds false.

A floating point type is one of **float**, **double**, or **long double**.

Example

```
// std_type_traits_is_floating_point.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_floating_point<trivial> == " << std::boolalpha
        << std::is_floating_point<trivial>::value << std::endl;
    std::cout << "is_floating_point<int> == " << std::boolalpha
        << std::is_floating_point<int>::value << std::endl;
    std::cout << "is_floating_point<float> == " << std::boolalpha
        << std::is_floating_point<float>::value << std::endl;

    return (0);
}
```

```
is_floating_point<trivial> == false
is_floating_point<int> == false
is_floating_point<float> == true
```

Requirements

Header: <type_traits>

Namespace: `std`

See also

[<type_traits>](#)

[is_integral](#) Class

is_function Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

Tests if type is a function type.

Syntax

```
template <class Ty>
struct is_function;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a function type, otherwise it holds false.

Example

```
// std__type_traits__is_function.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct functional
{
    int f();
};

int main()
{
    std::cout << "is_function<trivial> == " << std::boolalpha
        << std::is_function<trivial>::value << std::endl;
    std::cout << "is_function<functional> == " << std::boolalpha
        << std::is_function<functional>::value << std::endl;
    std::cout << "is_function<float()> == " << std::boolalpha
        << std::is_function<float()>::value << std::endl;

    return (0);
}
```

```
is_function<trivial> == false
is_function<functional> == false
is_function<float()> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_object Class](#)

is_fundamental Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is void or arithmetic.

Syntax

```
template <class Ty>
struct is_fundamental;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a fundamental type, that is, **void**, an integral type, an floating point type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

```
// std__type_traits__is_fundamental.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_fundamental<trivial> == " << std::boolalpha
        << std::is_fundamental<trivial>::value << std::endl;
    std::cout << "is_fundamental<int> == " << std::boolalpha
        << std::is_fundamental<int>::value << std::endl;
    std::cout << "is_fundamental<const float> == " << std::boolalpha
        << std::is_fundamental<const float>::value << std::endl;
    std::cout << "is_fundamental<void> == " << std::boolalpha
        << std::is_fundamental<void>::value << std::endl;

    return (0);
}
```

```
is_fundamental<trivial> == false
is_fundamental<int> == true
is_fundamental<const float> == true
is_fundamental<void> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_compound Class](#)

is_integral Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is integral.

Syntax

```
template <class Ty>
struct is_integral;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is one of the integral types, or a `cv-qualified` form of one of the integral types, otherwise it holds false.

An integral type is one of **bool**, **char**, **unsigned char**, **signed char**, **wchar_t**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, and **unsigned long**. In addition, with compilers that provide them, an integral type can be one of **long long**, **unsigned long long**, **__int64**, and **unsigned __int64**.

Example

```
// std__type_traits__is_integral.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_integral<trivial> == " << std::boolalpha
        << std::is_integral<trivial>::value << std::endl;
    std::cout << "is_integral<int> == " << std::boolalpha
        << std::is_integral<int>::value << std::endl;
    std::cout << "is_integral<float> == " << std::boolalpha
        << std::is_integral<float>::value << std::endl;

    return (0);
}
```

```
is_integral<trivial> == false
is_integral<int> == true
is_integral<float> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_enum](#) Class

[is_floating_point](#) Class

is_invocable, is_invocable_r, is_nothrow_invocable, is_nothrow_invocable_r classes

2/28/2019 • 2 minutes to read • [Edit Online](#)

These templates determine if a type can be invoked with the specified argument types. `is_invocable_r` and `is_nothrow_invocable_r` also determine if the result of the invocation is convertible to a specific type. `is_nothrow_invocable` and `is_nothrow_invocable_r` also determine if the invocation is known not to throw exceptions. Added in C++17.

Syntax

```
template <class Callable, class... Args>
struct is_invocable;

template <class Convertible, class Callable, class... Args>
struct is_invocable_r;

template <class Callable, class... Args>
struct is_nothrow_invocable;

template <class Convertible, class Callable, class... Args>
struct is_nothrow_invocable_r;

// Helper templates
template <class Callable, class... Args>
inline constexpr bool is_invocable_v =
    std::is_invocable<Callable, Args...>::value;

template <class Convertible, class Callable, class... Args>
inline constexpr bool is_invocable_r_v =
    std::is_invocable_r<Convertible, Callable, Args...>::value;

template <class Callable, class... Args>
inline constexpr bool is_nothrow_invocable_v =
    std::is_nothrow_invocable<Callable, Args...>::value;

template <class Convertible, class Callable, class... Args>
inline constexpr bool is_nothrow_invocable_r_v =
    std::is_nothrow_invocable_r<Convertible, Callable, Args...>::value;
```

Parameters

Callable

The callable type to query.

Args

The argument types to query.

Convertible

The type the result of *Callable* must be convertible to.

Remarks

The `is_invocable` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context.

The `is_invocable_r` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context to produce a result type convertible to *Convertible*.

The `is_nothrow_invocable` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context, and that such a call is known not to throw an exception.

The `is_nothrow_invocable_r` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context to produce a result type convertible to *Convertible*, and that such a call is known not to throw an exception.

Each of the types *Convertible*, *Callable*, and the types in the parameter pack *Args* must be a complete type, an array of unknown bound, or a possibly cv-qualified **void**. Otherwise, the behavior of the predicate is undefined.

Example

```
// std__type_traits__is_invocable.cpp
// compile using: cl /EHsc /std:c++17 std__type_traits__is_invocable.cpp
#include <type_traits>

auto test1(int) noexcept -> int (*)()
{
    return nullptr;
}

auto test2(int) -> int (*)()
{
    return nullptr;
}

int main()
{
    static_assert( std::is_invocable<decltype(test1), short>::value );

    static_assert( std::is_invocable_r<int(*)(), decltype(test1), int>::value );
    static_assert( std::is_invocable_r<long(*)(), decltype(test1), int>::value ); // fails

    static_assert( std::is_nothrow_invocable<decltype(test1), int>::value );
    static_assert( std::is_nothrow_invocable<decltype(test2), int>::value ); // fails

    static_assert( std::is_nothrow_invocable_r<int(*)(), decltype(test1), int>::value );
    static_assert( std::is_nothrow_invocable_r<int(*)(), decltype(test2), int>::value ); // fails
}
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[`<type_traits>`](#)

[invoke](#)

is_literal_type Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a type can be used as a `constexpr` variable or be constructed, used by, or returned from `constexpr` functions.

Syntax

```
template <class T>
struct is_literal_type;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a *literal type*, otherwise it holds false. A literal type is either **void**, a scalar type, a reference type, an array of literal type, or a literal class type. A literal class type is a class type that has a trivial destructor, is either an aggregate type or has at least one non-move, non-copy `constexpr` constructor, and all of its base classes and non-static data members are non-volatile literal types. While the type of a literal is always a literal type, the concept of a literal type includes anything that the compiler can evaluate as a `constexpr` at compile time.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_lvalue_reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is an lvalue reference.

Syntax

```
template <class Ty>
struct is_lvalue_reference;
```

Parameters

Ty

The type to query.

Remarks

An instance of this type predicate holds true if the type *Ty* is a reference to an object or to a function, otherwise it holds false. Note that *Ty* may not be an rvalue reference. For more information about rvalues, see [Rvalue Reference Declarator: &&](#).

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[Lvalues and Rvalues](#)

is_member_function_pointer Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a pointer to member function.

Syntax

```
template <class Ty>
struct is_member_function_pointer;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member function or a `cv-qualified` pointer to member function, otherwise it holds false.

Example

```
// std_type_traits__is_member_function_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct functional
{
    int f();
};

int main()
{
    std::cout << "is_member_function_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_function_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_function_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_function_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_function_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_function_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
}
```

```
is_member_function_pointer<trivial *> == false  
is_member_function_pointer<int trivial::*> == false  
is_member_function_pointer<int (functional::*)()> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_member_pointer](#) Class

is_member_object_pointer Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a pointer to member object.

Syntax

```
template <class Ty>
struct is_member_object_pointer;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member object or a `cv-qualified` pointer to member object, otherwise it holds false. Note that `is_member_object_pointer` holds false if *Ty* is a pointer to member function.

Example

```
// std__type_traits__is_member_object_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct functional
{
    int f();
};

int main()
{
    std::cout << "is_member_object_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_object_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_object_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_object_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_object_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_object_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
}
```

```
is_member_object_pointer<trivial *> == false  
is_member_object_pointer<int trivial::*> == true  
is_member_object_pointer<int (functional::*)()> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_member_pointer](#) Class

is_member_pointer Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a pointer to member.

Syntax

```
template <class Ty>
struct is_member_pointer;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member function or a pointer to member object, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

```
// std_type_traits_is_member_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct functional
{
    int f();
};

int main()
{
    std::cout << "is_member_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
}
```

```
is_member_pointer<trivial *> == false  
is_member_pointer<int trivial::*> == true  
is_member_pointer<int (functional::*)()> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_member_function_pointer](#) Class

[is_member_object_pointer](#) Class

[is_pointer](#) Class

is_move_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if the type can be move assigned.

Syntax

```
template <class T>
struct is_move_assignable;
```

Parameters

T

The type to query.

Remarks

A type is move assignable if an rvalue reference to the type can be assigned to a reference to the type. The type predicate is equivalent to `is_assignable<T&, T&&>`. Move assignable types include referenceable scalar types and class types that have either compiler-generated or user-defined move assignment operators.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_move_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type has a move constructor.

Syntax

```
template <class T>
struct is_move_constructible;
```

Parameters

T

The type to be evaluated

Remarks

A type predicate that evaluates to true if the type *T* can be constructed by using a move operation. This predicate is equivalent to `is_constructible<T, T&&>`.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a value of *From* type can be assigned to *To* type and the assignment is known not to throw.

Syntax

```
template <class To, class From>
struct is_nothrow_assignable;
```

Parameters

To

The type of the object that receives the assignment.

From

The type of the object that provides the value.

Remarks

The expression `declval<To>() = declval<From>()` must be well-formed and must be known to the compiler not to throw. Both *From* and *To* must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a type is constructible and is known not to throw when the specified argument types are used.

Syntax

```
template <class T, class... Args>
struct is_nothrow_constructible;
```

Parameters

T

The type to query.

Args

The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is constructible by using the argument types in *Args*, and the constructor is known by the compiler not to throw; otherwise it holds false. Type *T* is constructible if the variable definition `T t(std::declval<Args>()...);` is well-formed. Both *T* and all the types in *Args* must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_copy_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has a copy assignment operator that is known to the compiler not to throw.

Syntax

```
template <class T>
struct is_nothrow_copy_assignable;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true for a referenceable type *T* where `is_nothrow_assignable<T&, const T&>` holds true; otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[is_nothrow_assignable Class](#)

is_nothrow_copy_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has a **nothrow** copy constructor.

Syntax

```
template <class Ty>
struct is_nothrow_copy_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow copy constructor, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_default_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has a non-throwing default constructor.

Syntax

```
template <class Ty>
struct is_nothrow_default_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow default constructor, otherwise it holds false. An instance of the type predicate is equivalent to `is_nothrow_constructible<Ty>`.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_destructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is destructible and the destructor is known to the compiler not to throw.

Syntax

```
template <class T>  
struct is_nothrow_destructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, and the destructor is known to the compiler not to throw. Otherwise, it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_move_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has a **nothrow** move assignment operator.

Syntax

```
template <class Ty>
struct is_nothrow_move_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow move assignment operator, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_nothrow_move_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether type has a **nothrow** move constructor.

Syntax

```
template <class Ty>
struct is_nothrow_move_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow move constructor, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_null_pointer Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is `std::nullptr_t`.

Syntax

```
template <class T>
struct is_null_pointer;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is `std::nullptr_t`, otherwise it holds false.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_object Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is an object type.

Syntax

```
template <class Ty>
struct is_object;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds false if the type *Ty* is a reference type, a function type, or void, or a `cv-qualified` form of one of them, otherwise holds true.

Example

```
// std__type_traits__is_object.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct functional
{
    int f();
};

int main()
{
    std::cout << "is_object<trivial> == " << std::boolalpha
        << std::is_object<trivial>::value << std::endl;
    std::cout << "is_object<functional> == " << std::boolalpha
        << std::is_object<functional>::value << std::endl;
    std::cout << "is_object<trivial&> == " << std::boolalpha
        << std::is_object<trivial&>::value << std::endl;
    std::cout << "is_object<float()> == " << std::boolalpha
        << std::is_object<float()>::value << std::endl;
    std::cout << "is_object<void> == " << std::boolalpha
        << std::is_object<void>::value << std::endl;

    return (0);
}
```

```
is_object<trivial> == true  
is_object<functional> == true  
is_object<trivial&> == false  
is_object<float()> == false  
is_object<void> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_function](#) Class

is_pod Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is POD.

Syntax

```
template <class T>
struct is_pod;
```

Parameters

T

The type to query.

Remarks

`is_pod<T>::value` is **true** if the type *T* is Plain Old Data (POD). Otherwise it is **false**.

Arithmetic types, enumeration types, pointer types, and pointer to member types are POD.

A cv-qualified version of a POD type is itself a POD type.

An array of POD is itself POD.

A struct or union, all of whose non-static data members are POD, is itself POD if it has:

- No user-declared constructors.
- No private or protected non-static data members.
- No base classes.
- No virtual functions.
- No non-static data members of reference type.
- No user-defined copy assignment operator.
- No user-defined destructor.

Therefore, you can recursively build POD structs and arrays that contain POD structs and arrays.

Example

```

// std__type_traits__is_pod.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial {
    int val;
};

struct throws {
    throws() {} // User-declared ctor, so not POD

    int val;
};

int main() {
    std::cout << "is_pod<trivial> == " << std::boolalpha
        << std::is_pod<trivial>::value << std::endl;
    std::cout << "is_pod<int> == " << std::boolalpha
        << std::is_pod<int>::value << std::endl;
    std::cout << "is_pod<throws> == " << std::boolalpha
        << std::is_pod<throws>::value << std::endl;

    return (0);
}

```

```

is_pod<trivial> == true
is_pod<int> == true
is_pod<throws> == false

```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_pointer Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a pointer.

Syntax

```
template <class Ty>
struct is_pointer;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to **void**, a pointer to an object, or a pointer to a function, or a `cv-qualified` form of one of them, otherwise it holds false. Note that `is_pointer` holds false if *Ty* is a pointer to member or a pointer to member function.

Example

```
// std__type_traits__is_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_pointer<trivial> == " << std::boolalpha
        << std::is_pointer<trivial>::value << std::endl;
    std::cout << "is_pointer<int trivial::*> == " << std::boolalpha
        << std::is_pointer<int trivial::*>::value << std::endl;
    std::cout << "is_pointer<trivial *> == " << std::boolalpha
        << std::is_pointer<trivial *>::value << std::endl;
    std::cout << "is_pointer<int> == " << std::boolalpha
        << std::is_pointer<int>::value << std::endl;
    std::cout << "is_pointer<int *> == " << std::boolalpha
        << std::is_pointer<int *>::value << std::endl;

    return (0);
}
```

```
is_pointer<trivial> == false
is_pointer<int trivial::*> == false
is_pointer<trivial *> == true
is_pointer<int> == false
is_pointer<int *> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_member_pointer](#) Class

[is_reference](#) Class

is_polymorphic Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type has a virtual function.

Syntax

```
template <class Ty>  
struct is_polymorphic;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that declares or inherits a virtual function, otherwise it holds false.

Example

```

// std__type_traits__is_polymorphic.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

struct throws
{
    throws() throw(int)
    {
    }

    throws(const throws&) throw(int)
    {
    }

    throws& operator=(const throws&) throw(int)
    {
    }

    virtual ~throws()
    {
    }

    int val;
};

int main()
{
    std::cout << "is_polymorphic<trivial> == " << std::boolalpha
        << std::is_polymorphic<trivial>::value << std::endl;
    std::cout << "is_polymorphic<throws> == " << std::boolalpha
        << std::is_polymorphic<throws>::value << std::endl;

    return (0);
}

```

```

is_polymorphic<trivial> == false
is_polymorphic<throws> == true

```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_abstract Class](#)

is_reference Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a reference.

Syntax

```
template <class Ty>
struct is_reference;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a reference to an object or to a function, otherwise it holds false.

Example

```
// std__type_traits__is_reference.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_reference<trivial> == " << std::boolalpha
        << std::is_reference<trivial>::value << std::endl;
    std::cout << "is_reference<trivial&> == " << std::boolalpha
        << std::is_reference<trivial&>::value << std::endl;
    std::cout << "is_reference<int()> == " << std::boolalpha
        << std::is_reference<int()>::value << std::endl;
    std::cout << "is_reference<int(&)()> == " << std::boolalpha
        << std::is_reference<int(&)()>::value << std::endl;

    return (0);
}
```

```
is_reference<trivial> == false
is_reference<trivial&> == true
is_reference<int()> == false
is_reference<int(&)()> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_pointer](#) Class

is_rvalue_reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is an rvalue reference.

Syntax

```
template <class Ty>
struct is_rvalue_reference;
```

Parameters

Ty

The type to query.

Remarks

An instance of this type predicate holds true if the type *Ty* is an [rvalue reference](#).

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[Lvalues and Rvalues](#)

is_same Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if two types are the same.

Syntax

```
template <class Ty1, class Ty2>
struct is_same;
```

Parameters

Ty1

The first type to query.

Ty2

The second type to query.

Remarks

An instance of the type predicate holds true if the types *Ty1* and *Ty2* are the same type, otherwise it holds false.

Example

```
// std__type_traits__is_same.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct base
{
    int val;
};

struct derived
: public base
{
};

int main()
{
    std::cout << "is_same<base, base> == " << std::boolalpha
        << std::is_same<base, base>::value << std::endl;
    std::cout << "is_same<base, derived> == " << std::boolalpha
        << std::is_same<base, derived>::value << std::endl;
    std::cout << "is_same<derived, base> == " << std::boolalpha
        << std::is_same<derived, base>::value << std::endl;
    std::cout << "is_same<int, int> == " << std::boolalpha
        << std::is_same<int, int>::value << std::endl;
    std::cout << "is_same<int, const int> == " << std::boolalpha
        << std::is_same<int, const int>::value << std::endl;

    return (0);
}
```

```
is_same<base, base> == true  
is_same<base, derived> == false  
is_same<derived, base> == false  
is_same<int, int> == true  
is_same<int, const int> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_convertible](#) Class

[is_base_of](#) Class

is_scalar Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is scalar.

Syntax

```
template <class Ty>
struct is_scalar;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an integral type, a floating point type, an enumeration type, a pointer type, or a pointer to member type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

```
// std__type_traits__is_scalar.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_scalar<trivial> == " << std::boolalpha
        << std::is_scalar<trivial>::value << std::endl;
    std::cout << "is_scalar<trivial*> == " << std::boolalpha
        << std::is_scalar<trivial*>::value << std::endl;
    std::cout << "is_scalar<int> == " << std::boolalpha
        << std::is_scalar<int>::value << std::endl;
    std::cout << "is_scalar<float> == " << std::boolalpha
        << std::is_scalar<float>::value << std::endl;

    return (0);
}
```

```
is_scalar<trivial> == false
is_scalar<trivial*> == true
is_scalar<int> == true
is_scalar<float> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_compound](#) Class

is_signed Class

4/12/2019 • 2 minutes to read • [Edit Online](#)

Test if type is signed integer.

Syntax

```
template <class Ty>
struct is_signed;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a signed integral type or a `cv-qualified` signed integral type, otherwise it holds false.

Example

```
// std__type_traits__is_signed.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_signed<trivial> == " << std::boolalpha
        << std::is_signed<trivial>::value << std::endl;
    std::cout << "is_signed<int> == " << std::boolalpha
        << std::is_signed<int>::value << std::endl;
    std::cout << "is_signed<unsigned int> == " << std::boolalpha
        << std::is_signed<unsigned int>::value << std::endl;
    std::cout << "is_signed<float> == " << std::boolalpha
        << std::is_signed<float>::value << std::endl;

    return (0);
}
```

```
is_signed<trivial> == false
is_signed<int> == true
is_signed<unsigned int> == false
is_signed<float> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_unsigned Class](#)

is_standard_layout Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a standard layout.

Syntax

```
template <class Ty>
struct is_standard_layout;
```

Parameters

PARAMETER	DESCRIPTION
<i>Ty</i>	The type to query

Remarks

An instance of this type predicate holds true if the type *Ty* is a class that has a standard layout of member objects in memory, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivial Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is a trivial type.

Syntax

```
template <class T>  
struct is_trivial;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a trivial type, otherwise it holds false. Trivial types are scalar types, trivially copyable class types, arrays of these types and cv-qualified versions of these types.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a value of `From` type can be trivially assigned to `To` type

Syntax

```
template <class To, class From>
struct is_trivially_assignable;
```

Parameters

To

The type of the object that receives the assignment.

From

The type of the object that provides the value.

Remarks

The expression `declval<To>() = declval<From>()` must be well-formed, and must be known to the compiler to require no non-trivial operations. Both `From` and `To` must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

is_trivially_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether a type is trivially constructible when the specified argument types are used.

Syntax

```
template <class T, class... Args>
struct is_trivially_constructible;
```

Parameters

T

The type to query.

Args

The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is trivially constructible by using the argument types in *Args*, otherwise it holds false. Type *T* is trivially constructible if the variable definition

`T t(std::declval<Args>()...);` is well-formed and is known to call no non-trivial operations. Both *T* and all the types in *Args* must be complete types, **void**, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_copy_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type has a trivial copy assignment operator.

Syntax

```
template <class Ty>
struct is_trivially_copy_assignable;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class that has a trivial copy assignment operator, otherwise it holds false.

An assignment constructor for a class *T* is trivial if it is implicitly provided, the class *T* has no virtual functions, the class *T* has no virtual bases, the classes of all the non-static data members of class type have trivial assignment operators, and the classes of all the non-static data members of type array of class have trivial assignment operators.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_copy_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if the type has a trivial copy constructor.

Syntax

```
template <class T>
struct is_trivially_copy_constructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class that has a trivial copy constructor, otherwise it holds false.

A copy constructor for a class *T* is trivial if it is implicitly declared, the class *T* has no virtual functions or virtual bases, all the direct bases of class *T* have trivial copy constructors, the classes of all the non-static data members of class type have trivial copy constructors, and the classes of all the non-static data members of type array of class have trivial copy constructors.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_copyable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is a trivially copyable type.

Syntax

```
template <class T>  
struct is_trivially_copyable;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a trivially copyable type, otherwise it holds false. Trivially copyable types have no non-trivial copy operations, move operations, or destructors. Generally, a copy operation is considered trivial if it can be implemented as a bitwise copy. Both built-in types and arrays of trivially copyable types are trivially copyable.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_default_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type has trivial default constructor.

Syntax

```
template <class Ty>
struct is_trivially_default_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial constructor, otherwise it holds false.

A default constructor for a class *Ty* is trivial if:

- it is an implicitly declared default constructor
- the class *Ty* has no virtual functions
- the class *Ty* has no virtual bases
- all the direct bases of the class *Ty* have trivial constructors
- the classes of all the non-static data members of class type have trivial constructors
- the classes of all the non-static data members of type array of class have trivial constructors

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_destructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is trivially destructible.

Syntax

```
template <class T>
struct is_trivially_destructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, and the destructor is known to the compiler to use no non-trivial operations. Otherwise, it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_move_assignable Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type has a trivial move assignment operator.

Syntax

```
template <class Ty>
struct is_trivially_move_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move assignment operator, otherwise it holds false.

A move assignment operator for a class *Ty* is trivial if:

it is implicitly provided

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the classes of all the non-static data members of class type have trivial move assignment operators

the classes of all the non-static data members of type array of class have trivial move assignment operators

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_trivially_move_constructible Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Tests if type has trivial move constructor.

Syntax

```
template <class Ty>
struct is_trivially_move_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move constructor, otherwise it holds false.

A move constructor for a class *Ty* is trivial if:

it is implicitly declared

its parameter types are equivalent to those of an implicit declaration

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the class has no volatile non-static data members

all the direct bases of the class *Ty* have trivial move constructors

the classes of all the non-static data members of class type have trivial move constructors

the classes of all the non-static data members of type array of class have trivial move constructors

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_union Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is a union.

Syntax

```
template <class Ty>
struct is_union;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a union type or a `cv-qualified` form of a union type, otherwise it holds false.

Example

```
// std__type_traits__is_union.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

union ints
{
    int in;
    long lo;
};

int main()
{
    std::cout << "is_union<trivial> == " << std::boolalpha
        << std::is_union<trivial>::value << std::endl;
    std::cout << "is_union<int> == " << std::boolalpha
        << std::is_union<int>::value << std::endl;
    std::cout << "is_union<ints> == " << std::boolalpha
        << std::is_union<ints>::value << std::endl;

    return (0);
}
```

```
is_union<trivial> == false
is_union<int> == false
is_union<ints> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_class](#) [Class](#)

is_unsigned Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is unsigned integer.

Syntax

```
template <class Ty>
struct is_unsigned;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an unsigned integral type or a `cv-qualified` unsigned integral type, otherwise it holds false.

Example

```
// std__type_traits__is_unsigned.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_unsigned<trivial> == " << std::boolalpha
        << std::is_unsigned<trivial>::value << std::endl;
    std::cout << "is_unsigned<int> == " << std::boolalpha
        << std::is_unsigned<int>::value << std::endl;
    std::cout << "is_unsigned<unsigned int> == " << std::boolalpha
        << std::is_unsigned<unsigned int>::value << std::endl;
    std::cout << "is_unsigned<float> == " << std::boolalpha
        << std::is_unsigned<float>::value << std::endl;

    return (0);
}
```

```
is_unsigned<trivial> == false
is_unsigned<int> == false
is_unsigned<unsigned int> == true
is_unsigned<float> == false
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_signed Class](#)

is_void Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests whether the type is void.

Syntax

```
template <class T>
struct is_void;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is **void** or a cv-qualified form of **void**, otherwise it holds false.

Example

```
// std_type_traits_is_void.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_void<trivial> == " << std::boolalpha
        << std::is_void<trivial>::value << std::endl;
    std::cout << "is_void<void()> == " << std::boolalpha
        << std::is_void<void()>::value << std::endl;
    std::cout << "is_void<void> == " << std::boolalpha
        << std::is_void<void>::value << std::endl;

    return (0);
}
```

```
is_void<trivial> == false
is_void<void()> == false
is_void<void> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

is_volatile Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Tests if type is volatile.

Syntax

```
template <class Ty>
struct is_volatile;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if *Ty* is `volatile-qualified`.

Example

```
// std__type_traits__is_volatile.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_volatile<trivial> == " << std::boolalpha
        << std::is_volatile<trivial>::value << std::endl;
    std::cout << "is_volatile<volatile trivial> == " << std::boolalpha
        << std::is_volatile<volatile trivial>::value << std::endl;
    std::cout << "is_volatile<int> == " << std::boolalpha
        << std::is_volatile<int>::value << std::endl;
    std::cout << "is_volatile<volatile int> == " << std::boolalpha
        << std::is_volatile<volatile int>::value << std::endl;

    return (0);
}
```

```
is_volatile<trivial> == false
is_volatile<volatile trivial> == true
is_volatile<int> == false
is_volatile<volatile int> == true
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[is_const](#) [Class](#)

make_signed Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes type or the smallest signed type greater than or equal in size to type.

Syntax

```
template <class T>
struct make_signed;

template <class T>
using make_signed_t = typename make_signed<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if `is_signed<T>` holds true. Otherwise it is the smallest unsigned type `UT` for which `sizeof (T) <= sizeof (UT)` .

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

make_unsigned Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes type or the smallest unsigned type greater than or equal in size to type.

Syntax

```
template <class T>
struct make_unsigned;

template <class T>
using make_unsigned_t = typename make_unsigned<T>::type;
```

Parameters

PARAMETER	DESCRIPTION
<i>T</i>	The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if `is_unsigned<T>` holds true. Otherwise it is the smallest signed type `ST` for which `sizeof (T) <= sizeof (ST)` .

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

rank Class

11/9/2018 • 2 minutes to read • [Edit Online](#)

Gets number of array dimensions.

Syntax

```
template <class Ty>
struct rank;
```

Parameters

Ty

The type to query.

Remarks

The type query holds the value of the number of dimensions of the array type *Ty*, or 0 if *Ty* is not an array type.

Example

```
// std__type_traits__rank.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << "rank<int> == "
        << std::rank<int>::value << std::endl;
    std::cout << "rank<int[5]> == "
        << std::rank<int[5]>::value << std::endl;
    std::cout << "rank<int[5][10]> == "
        << std::rank<int[5][10]>::value << std::endl;

    return (0);
}
```

```
rank<int> == 0
rank<int[5]> == 1
rank<int[5][10]> == 2
```

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[extent Class](#)

remove_all_extents Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes non array type from array type.

Syntax

```
template <class T>
struct remove_all_extents;

template <class T>
using remove_all_extents_t = typename remove_all_extents<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_all_extents<T>` holds a modified-type that is the element type of the array type *T* with all array dimensions removed, or *T* if *T* is not an array type.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << "remove_all_extents<int> == "
        << typeid(std::remove_all_extents_t<int>).name()
        << std::endl;
    std::cout << "remove_all_extents_t<int[5]> == "
        << typeid(std::remove_all_extents_t<int[5]>).name()
        << std::endl;
    std::cout << "remove_all_extents_t<int[5][10]> == "
        << typeid(std::remove_all_extents_t<int[5][10]>).name()
        << std::endl;

    return (0);
}
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_extent Class](#)

remove_const Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes a non const type from type.

Syntax

```
template <class T>
struct remove_const;
```

```
template <class T>
using remove_const_t = typename remove_const<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_const<T>` holds a modified-type that is `T1` when *T* is of the form `const T1`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    int *p = (std::remove_const_t<const int>*)0;

    p = p; // to quiet "unused" warning
    std::cout << "remove_const_t<const int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
remove_const_t<const int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[add_const Class](#)

remove_cv Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes non const/volatile type from type.

Syntax

```
template <class T>
struct remove_cv;

template <class T>
using remove_cv_t = typename remove_cv<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_cv<T>` holds a modified-type that is `T1` when *T* is of the form `const T1`, `volatile T1`, or `const volatile T1`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    int *p = (std::remove_cv_t<const volatile int> *)0;

    p = p; // to quiet "unused" warning
    std::cout << "remove_cv_t<const volatile int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
remove_cv_t<const volatile int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[remove_const Class](#)

remove_volatile Class

remove_extent Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes element type from array type.

Syntax

```
template <class T>
struct remove_extent;

template <class T>
using remove_extent_t = typename remove_extent<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_extent<T>` holds a modified-type that is `T1` when *T* is of the form `T1[N]`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << "remove_extent_t<int> == "
              << typeid(std::remove_extent_t<int>).name()
              << std::endl;
    std::cout << "remove_extent_t<int[5]> == "
              << typeid(std::remove_extent_t<int[5]>).name()
              << std::endl;
    std::cout << "remove_extent_t<int[5][10]> == "
              << typeid(std::remove_extent_t<int[5][10]>).name()
              << std::endl;
    return (0);
}
```

```
remove_extent_t<int> == int
remove_extent_t<int[5]> == int
remove_extent_t<int[5][10]> == int [10]
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

`<type_traits>`

`remove_all_extents` Class

remove_pointer Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes type from pointer to type.

Syntax

```
template <class T>
struct remove_pointer;

template <class T>
using remove_pointer_t = typename remove_pointer<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_pointer<T>` holds a modified-type that is `T1` when *T* is of the form `T1*`, `T1* const`, `T1* volatile`, or `T1* const volatile`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    int *p = (std::remove_pointer_t<int *> *)0;

    p = p; // to quiet "unused" warning
    std::cout << "remove_pointer_t<int *> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
remove_pointer_t<int *> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[add_pointer Class](#)

remove_reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes non reference type from type.

Syntax

```
template <class T>
struct remove_reference;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_reference<T>` holds a modified-type that is `T1` when *T* is of the form `T1&`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    int *p = (std::remove_reference_t<int&> *)0;

    p = p; // to quiet "unused" warning
    std::cout << "remove_reference_t<int&> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
remove_reference_t<int&> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[add_lvalue_reference Class](#)

remove_volatile Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Makes non volatile type from type.

Syntax

```
template <class T>
struct remove_volatile;

template <class T>
using remove_volatile_t = typename remove_volatile<T>::type;
```

Parameters

T

The type to modify.

Remarks

An instance of `remove_volatile<T>` holds a modified-type that is `T1` when *T* is of the form `volatile T1`, otherwise *T*.

Example

```
#include <type_traits>
#include <iostream>

int main()
{
    int *p = (std::remove_volatile_t<volatile int> *)0;

    p = p; // to quiet "unused" warning
    std::cout << " remove_volatile_t<volatile int> == "
              << typeid(*p).name() << std::endl;

    return (0);
}
```

```
remove_volatile_t<volatile int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

[add_volatile Class](#)

result_of Class

2/28/2019 • 2 minutes to read • [Edit Online](#)

Determines the return type of the callable type that takes the specified argument types. Added in C++14, deprecated in C++17.

Syntax

```
template<class>
struct result_of; // Causes a static assert

template <class Fn, class... ArgTypes>
struct result_of<Fn(ArgTypes...)>;

// Helper type
template<class T>
    using result_of_t = typename result_of<T>::type;
```

Parameters

Fn

The callable type to query.

ArgTypes

The types of the argument list to the callable type to query.

Remarks

Use this template to determine at compile time the result type of `Fn (ArgTypes)`, where *Fn* is a callable type, reference to function, or reference to callable type, invoked using an argument list of the types in *ArgTypes*. The

`type` member of the template class names the result type of

`decltype(std::invoke(declval<Fn>(), declval<ArgTypes>()...))` if the unevaluated expression

`std::invoke(declval<Fn>(), declval<ArgTypes>()...)` is well-formed. Otherwise, the template class has no member

`type`. The type *Fn* and all types in the parameter pack *ArgTypes* must be complete types, **void**, or arrays of unknown bound. Deprecated in favor of [invoke_result](#) in C++17.

Requirements

Header: <type_traits>

Namespace: std

See also

[<type_traits>](#)

[invoke_result class](#)

underlying_type Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

Produces the underlying integral type for an enumeration type.

Syntax

```
template <class T>
struct underlying_type;
```

Parameters

T

The type to modify.

Remarks

The `type` member typedef of the template class names the underlying integral type of *T*, when *T* is an enumeration type, otherwise there is no member typedef `type`.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

[<type_traits>](#)

<type_traits> functions

11/9/2018 • 4 minutes to read • [Edit Online](#)

is_assignable	is_copy_assignable	is_copy_constructible
is_default_constructible	is_move_assignable	is_move_constructible
is_nothrow_move_assignable	is_trivially_copy_assignable	is_trivially_move_assignable
is_trivially_move_constructible		

is_assignable

Tests whether a value of *From* type can be assigned to a *To* type.

```
template <class To, class From>
struct is_assignable;
```

Parameters

To

The type of the object that receives the assignment.

From

The type of the object that provides the value.

Remarks

The unevaluated expression `declval<To>() = declval<From>()` must be well-formed. Both *From* and *To* must be complete types, **void**, or arrays of unknown bound.

is_copy_assignable

Tests whether type has can be copied on assignment.

```
template <class Ty>
struct is_copy_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy assignment operator, otherwise it holds false. Equivalent to `is_assignable<Ty&, const Ty&>`.

is_copy_constructible

Tests if type has a copy constructor.

```
template <class Ty>
struct is_copy_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy constructor, otherwise it holds false.

Example

```
#include <type_traits>
#include <iostream>

struct Copyable
{
    int val;
};

struct NotCopyable
{
    NotCopyable(const NotCopyable&) = delete;
    int val;
};

int main()
{
    std::cout << "is_copy_constructible<Copyable> == " << std::boolalpha
        << std::is_copy_constructible<Copyable>::value << std::endl;
    std::cout << "is_copy_constructible<NotCopyable> == " << std::boolalpha
        << std::is_copy_constructible<NotCopyable>::value << std::endl;

    return (0);
}
```

```
is_copy_constructible<Copyable> == true
is_copy_constructible<NotCopyable> == false
```

is_default_constructible

Tests if a type has a default constructor.

```
template <class Ty>
struct is_default_constructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type that has a default constructor, otherwise it holds false. This is equivalent to the predicate `is_constructible<T>`. Type *T* must be a complete type, **void**, or an array of unknown bound.

Example

```
#include <type_traits>
#include <iostream>

struct Simple
{
    Simple() : val(0) {}
    int val;
};

struct Simple2
{
    Simple2(int v) : val(v) {}
    int val;
};

int main()
{
    std::cout << "is_default_constructible<Simple> == " << std::boolalpha
        << std::is_default_constructible<Simple>::value << std::endl;
    std::cout << "is_default_constructible<Simple2> == " << std::boolalpha
        << std::is_default_constructible<Simple2>::value << std::endl;

    return (0);
}
```

```
is_default_constructible<Simple> == true
is_default_constructible<Simple2> == false
```

is_move_assignable

Tests if the type can be move assigned.

```
template <class T>
struct is_move_assignable;
```

Parameters

T

The type to query.

Remarks

A type is move assignable if an rvalue reference to the type can be assigned to a reference to the type. The type predicate is equivalent to `is_assignable<T&, T&&>`. Move assignable types include referenceable scalar types and class types that have either compiler-generated or user-defined move assignment operators.

is_move_constructible

Tests whether the type has a move constructor.

```
template <class T>
struct is_move_constructible;
```

Parameters

T

The type to be evaluated

Remarks

A type predicate that evaluates to true if the type T can be constructed by using a move operation. This predicate is equivalent to `is_constructible<T, T&&>`.

is_nothrow_move_assignable

Tests whether type has a **nothrow** move assignment operator.

```
template <class Ty>
struct is_nothrow_move_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type Ty has a nothrow move assignment operator, otherwise it holds false.

is_trivially_copy_assignable

Tests whether the type has a trivial copy assignment operator.

```
template <class Ty>
struct is_trivially_copy_assignable;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type T is a class that has a trivial copy assignment operator, otherwise it holds false.

An assignment constructor for a class T is trivial if it is implicitly provided, the class T has no virtual functions, the class T has no virtual bases, the classes of all the non-static data members of class type have trivial assignment operators, and the classes of all the non-static data members of type array of class have trivial assignment operators.

is_trivially_move_assignable

Tests whether the type has a trivial move assignment operator.

```
template <class Ty>
struct is_trivially_move_assignable;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type Ty is a class that has a trivial move assignment operator,

otherwise it holds false.

A move assignment operator for a class *Ty* is trivial if:

it is implicitly provided

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the classes of all the non-static data members of class type have trivial move assignment operators

the classes of all the non-static data members of type array of class have trivial move assignment operators

is_trivially_move_constructible

Tests if type has trivial move constructor.

```
template <class Ty>
struct is_trivially_move_constructible;
```

Parameters

Ty

The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move constructor, otherwise it holds false.

A move constructor for a class *Ty* is trivial if:

it is implicitly declared

its parameter types are equivalent to those of an implicit declaration

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the class has no volatile non-static data members

all the direct bases of the class *Ty* have trivial move constructors

the classes of all the non-static data members of class type have trivial move constructors

the classes of all the non-static data members of type array of class have trivial move constructors

See also

[<type_traits>](#)

<type_traits> typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

false_type

true_type

false_type Typedef

Holds integral constant with false value.

```
typedef integral_constant<bool, false> false_type;
```

Remarks

The type is a synonym for a specialization of the template `integral_constant`.

Example

```
#include <type_traits>
#include <iostream>

int main() {
    std::cout << "false_type == " << std::boolalpha
               << std::false_type::value << std::endl;
    std::cout << "true_type == " << std::boolalpha
               << std::true_type::value << std::endl;

    return (0);
}
```

```
false_type == false
true_type == true
```

true_type Typedef

Holds integral constant with true value.

```
typedef integral_constant<bool, true> true_type;
```

Remarks

The type is a synonym for a specialization of the template `integral_constant`.

Example

```
// std__type_traits__true_type.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main() {
    std::cout << "false_type == " << std::boolalpha
        << std::false_type::value << std::endl;
    std::cout << "true_type == " << std::boolalpha
        << std::true_type::value << std::endl;

    return (0);
}
```

```
false_type == false
true_type == true
```

See also

[`<type_traits>`](#)

<typeindex>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Include the standard header <typeindex> to define a class and function that support the indexing of objects of class `type_info`.

Syntax

```
#include <typeindex>
```

Remarks

The [hash Structure](#) defines a `hash function` that's suitable for mapping values of type `type_index` to a distribution of index values.

The `type_index` class wraps a pointer to a `type_info` object to assist in indexing.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

hash Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

The template class defines its method as returning `val.hash_code()`. The method defines a hash function that is used to map values of type `type_index` to a distribution of index values.

Syntax

```
template <>
struct hash<type_index>
: public unary_function<type_index, size_t>
{ // hashes a typeinfo object
    size_t operator()(type_index val) const;

};
```

See also

[<typeindex>](#)

type_index Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `type_index` class wraps a pointer to [type_info Class](#) to assist in indexing by such objects.

```
class type_index { public: type_index(const type_info& tinfo); const char *name() const; size_t hash_code() const;
bool operator==(const type_info& right) const; bool operator!=(const type_info& right) const; bool
operator<(const type_info& right) const; bool operator<=(const type_info& right) const; bool operator>(const
type_info& right) const; bool operator>=(const type_info& right) const; };
```

The constructor initializes `ptr` to `&tinfo`.

`name` returns `ptr->name()`.

`hash_code` returns `ptr->hash_code()`.

`operator==` returns `*ptr == right.ptr`.

`operator!=` returns `!(*this == right)`.

`operator<` returns `*ptr->before(*right.ptr)`.

`operator<=` returns `!(right < *this)`.

`operator>` returns `right < *this`.

`operator>=` returns `!(*this < right)`.

See also

[Run-Time Type Information](#)

[<typeindex>](#)

<typeinfo>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Include the standard header <typeinfo> to define several types associated with the type-identification operator [typeid Operator](#), which yields information about both static and dynamic types.

Syntax

```
#include <typeinfo>
```

Remarks

For information on classes defined in <typeinfo>, see the following topics:

- [bad_cast Exception](#)
- [bad_typeid Exception](#)
- [type_info Class](#)

<unordered_map>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template classes [unordered_map](#) and [unordered_multimap](#) and their supporting templates.

Syntax

```
#include <unordered_map>
```

Classes

CLASS	DESCRIPTION
unordered_map Class	Stores hash table of {key, mapped} pairs.
unordered_multimap Class	Stores hash table of {key, mapped} pairs.

Functions

FUNCTION	DESCRIPTION
operator!=	Tests if the unordered_map object on the left side of the operator is not equal to the unordered_map object on the right side.
operator==	Tests if the unordered_map object on the left side of the operator is equal to the unordered_map object on the right side.
swap Function (unordered_map)	Swaps two maps.
operator!=	Tests if the unordered_multimap object on the left side of the operator is not equal to the unordered_multimap object on the right side.
operator==	Tests if the unordered_multimap object on the left side of the operator is equal to the unordered_multimap object on the right side.
swap Function (unordered_map)	Swaps two multimaps.

See also

[unordered_multiset Class](#)

[unordered_set Class](#)

<unordered_map> functions

11/9/2018 • 2 minutes to read • [Edit Online](#)

[swap \(unordered_map\)](#)

[swap \(unordered_multimap\)](#)

swap (unordered_map)

Swaps the contents of two containers.

```
template <class Key, class Ty, class Hash, class Pred, class Alloc>
void swap(
    unordered_map <Key, Ty, Hash, Pred, Alloc>& left,
    unordered_map <Key, Ty, Hash, Pred, Alloc>& right);
```

Parameters

Key

The key type.

Ty

The mapped type.

Hash

The hash function object type.

Pred

The equality comparison function object type.

Alloc

The allocator class.

left

The first container to swap.

right

The second container to swap.

Remarks

The template function executes `left.unordered_map::swap(right)`.

Example

```

// std__unordered_map__u_m_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

    // display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]

```

swap (unordered_multimap)

Swaps the contents of two containers.

```

template <class Key, class Ty, class Hash, class Pred, class Alloc>
void swap(
    unordered_multimap <Key, Ty, Hash, Pred, Alloc>& left,
    unordered_multimap <Key, Ty, Hash, Pred, Alloc>& right);

```

Parameters

Key

The key type.

Ty

The mapped type.

Hash

The hash function object type.

Pred

The equality comparison function object type.

Alloc

The allocator class.

left

The first container to swap.

right

The second container to swap.

Remarks

The template function executes `left. unordered_multimap::swap (right) .`

Example

```

// std__unordered_map__u__mm_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

    // display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]

```

See also

[<unordered_map>](#)

<unordered_map> operators

11/9/2018 • 4 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator==</code>	<code>operator!=</code>	<code>operator==</code>

operator!=

Tests whether the `unordered_map` object on the left side of the operator is not equal to the `unordered_map` object on the right side.

```
bool operator!=(const unordered_map <Key, Type, Hash, Pred, Allocator>& left, const unordered_map <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_map`.

right

An object of type `unordered_map`.

Return Value

true if the `unordered_maps` are not equal; **false** if they are equal.

Remarks

The comparison between `unordered_map` objects is not affected by the arbitrary order in which they store their elements. Two `unordered_maps` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```
// unordered_map_op_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
    using namespace std;
    unordered_map<int, int> um1, um2, um3;

    for ( int i = 0 ; i < 3 ; ++i ) {
        um1.insert( make_pair( i+1, i ) );
        um1.insert( make_pair( i, i ) );

        um2.insert( make_pair( i, i+1 ) );
        um2.insert( make_pair( i, i ) );

        um3.insert( make_pair( i, i ) );
        um3.insert( make_pair( i+1, i ) );
    }

    cout << boolalpha;
    cout << "um1 != um2: " << (um1 != um2) << endl;
    cout << "um1 != um3: " << (um1 != um3) << endl;
    cout << "um2 != um3: " << (um2 != um3) << endl;
}
```

Output:

```
um1 != um2: true
```

```
um1 != um3: false
```

```
um2 != um3: true
```

operator==

Tests whether the [unordered_map](#) object on the left side of the operator is equal to the `unordered_map` object on the right side.

```
bool operator==(const unordered_map <Key, Type, Hash, Pred, Allocator>& left, const unordered_map <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_map`.

right

An object of type `unordered_map`.

Return Value

true if the `unordered_maps` are equal; **false** if they are not equal.

Remarks

The comparison between `unordered_map` objects is not affected by the arbitrary order in which they store their elements. Two `unordered_maps` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```

// unordered_map_op_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
    using namespace std;
    unordered_map<int, int> um1, um2, um3;

    for ( int i = 0 ; i < 3 ; ++i ) {
        um1.insert( make_pair( i+1, i ) );
        um1.insert( make_pair( i, i ) );

        um2.insert( make_pair( i, i+1 ) );
        um2.insert( make_pair( i, i ) );

        um3.insert( make_pair( i, i ) );
        um3.insert( make_pair( i+1, i ) );
    }

    cout << boolalpha;
    cout << "um1 == um2: " << (um1 == um2) << endl;
    cout << "um1 == um3: " << (um1 == um3) << endl;
    cout << "um2 == um3: " << (um2 == um3) << endl;
}

```

Output:

```
um1 == um2: false
```

```
um1 == um3: true
```

```
um2 == um3: false
```

operator!=

Tests whether the [unordered_multimap](#) object on the left side of the operator is not equal to the [unordered_multimap](#) object on the right side.

```

bool operator!=(const unordered_multimap<Key, Type, Hash, Pred, Allocator>& left, const unordered_multimap<Key, Type, Hash, Pred, Allocator>& right);

```

Parameters

left

An object of type `unordered_multimap` .

right

An object of type `unordered_multimap` .

Return Value

true if the [unordered_multimaps](#) are not equal; **false** if they are equal.

Remarks

The comparison between [unordered_multimap](#) objects is not affected by the arbitrary order in which they store their elements. Two [unordered_multimaps](#) are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are not equal.

Example


```
// unordered_multimap_op_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
    using namespace std;
    unordered_multimap<int, int> um1, um2, um3;

    for ( int i = 0 ; i < 3 ; ++i ) {
        um1.insert( make_pair( i, i ) );
        um1.insert( make_pair( i, i ) );

        um2.insert( make_pair( i, i ) );
        um2.insert( make_pair( i, i ) );
        um2.insert( make_pair( i, i ) );

        um3.insert( make_pair( i, i ) );
        um3.insert( make_pair( i, i ) );
    }

    cout << boolalpha;
    cout << "um1 != um2: " << (um1 != um2) << endl;
    cout << "um1 != um3: " << (um1 != um3) << endl;
    cout << "um2 != um3: " << (um2 != um3) << endl;
}
```

Output:

```
um1 != um2: true
```

```
um1 != um3: false
```

```
um2 != um3: true
```

operator==

Tests whether the [unordered_multimap](#) object on the left side of the operator is equal to the `unordered_multimap` object on the right side.

```
bool operator==(const unordered_multimap <Key, Type, Hash, Pred, Allocator>& left, const unordered_multimap
<Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_multimap` .

right

An object of type `unordered_multimap` .

Return Value

true if the `unordered_multimaps` are equal; **false** if they are not equal.

Remarks

The comparison between `unordered_multimap` objects is not affected by the arbitrary order in which they store their elements. Two `unordered_multimaps` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```
// unordered_multimap_op_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
    using namespace std;
    unordered_multimap<int, int> um1, um2, um3;

    for ( int i = 0 ; i < 3 ; ++i ) {
        um1.insert( make_pair( i, i ) );
        um1.insert( make_pair( i, i ) );

        um2.insert( make_pair( i, i ) );
        um2.insert( make_pair( i, i ) );
        um2.insert( make_pair( i, i ) );

        um3.insert( make_pair( i, i ) );
        um3.insert( make_pair( i, i ) );
    }

    cout << boolalpha;
    cout << "um1 == um2: " << (um1 == um2) << endl;
    cout << "um1 == um3: " << (um1 == um3) << endl;
    cout << "um2 == um3: " << (um2 == um3) << endl;
}
```

Output:

```
um1 == um2: false
```

```
um1 == um3: true
```

```
um2 == um3: false
```

See also

[<unordered_map>](#)

unordered_map Class

11/9/2018 • 45 minutes to read • [Edit Online](#)

The template class describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element stores two objects, a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

```
template <class Key,  
          class Ty,  
          class Hash = std::hash<Key>,  
          class Pred = std::equal_to<Key>,  
          class Alloc = std::allocator<std::pair<const Key, Ty>>>  
class unordered_map;
```

Parameters

PARAMETER	DESCRIPTION
<i>Key</i>	The key type.
<i>Ty</i>	The mapped type.
<i>Hash</i>	The hash function object type.
<i>Pred</i>	The equality comparison function object type.
<i>Alloc</i>	The allocator class.

Members

TYPE DEFINITION	DESCRIPTION
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.

TYPE DEFINITION	DESCRIPTION
<code>const_pointer</code>	The type of a constant pointer to an element.
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>mapped_type</code>	The type of a mapped value associated with each key.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.
MEMBER FUNCTION	DESCRIPTION
<code>at</code>	Finds an element with the specified key.
<code>begin</code>	Designates the beginning of the controlled sequence.
<code>bucket</code>	Gets the bucket number for a key value.
<code>bucket_count</code>	Gets the number of buckets.
<code>bucket_size</code>	Gets the size of a bucket.
<code>cbegin</code>	Designates the beginning of the controlled sequence.
<code>cend</code>	Designates the end of the controlled sequence.
<code>clear</code>	Removes all elements.
<code>count</code>	Finds the number of elements matching a specified key.
<code>emplace</code>	Adds an element constructed in place.

MEMBER FUNCTION	DESCRIPTION
<code>emplace_hint</code>	Adds an element constructed in place, with hint.
<code>empty</code>	Tests whether no elements are present.
<code>end</code>	Designates the end of the controlled sequence.
<code>equal_range</code>	Finds range that matches a specified key.
<code>erase</code>	Removes elements at specified positions.
<code>find</code>	Finds an element that matches a specified key.
<code>get_allocator</code>	Gets the stored allocator object.
<code>hash_function</code>	Gets the stored hash function object.
<code>insert</code>	Adds elements.
<code>key_eq</code>	Gets the stored comparison function object.
<code>load_factor</code>	Counts the average elements per bucket.
<code>max_bucket_count</code>	Gets the maximum number of buckets.
<code>max_load_factor</code>	Gets or sets the maximum elements per bucket.
<code>max_size</code>	Gets the maximum size of the controlled sequence.
<code>rehash</code>	Rebuilds the hash table.
<code>size</code>	Counts the number of elements.
<code>swap</code>	Swaps the contents of two containers.
<code>unordered_map</code>	Constructs a container object.

OPERATOR	DESCRIPTION
<code>unordered_map::operator[]</code>	Finds or inserts an element with the specified key.
<code>unordered_map::operator=</code>	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `unordered_map::key_equal` and a hash function object of type `unordered_map::hasher`. You access the first stored object by calling the member function `unordered_map::key_eq()`; and you access the second stored object by calling the member function `unordered_map::hash_function()`. Specifically, for all values `x` and `y` of type `key`, the call `key_eq()(x, y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of

values of type `size_t`. Unlike template class [unordered_multimap Class](#), an object of template class `unordered_map` ensures that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys are unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `unordered_map::load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type `unordered_map::allocator_type`. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: `<unordered_map>`

Namespace: `std`

`unordered_map::allocator_type`

The type of an allocator for managing storage.

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

```
// std__unordered_map__unordered_map_allocator_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

unordered_map::at

Finds an element in a unordered_map with a specified key value.

```
Ty& at(const Key& key);  
const Ty& at(const Key& key) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>key</i>	The key value to find.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value is not found, then the function throws an object of class `out_of_range`.

Example

```
// unordered_map_at.cpp  
// compile with: /EHsc  
#include <unordered_map>  
#include <iostream>  
  
typedef std::unordered_map<char, int> Mymap;  
typedef std::unordered_map<char, int> Mymap;  
int main()  
{  
    Mymap c1;  
  
    c1.insert(Mymap::value_type('a', 1));  
    c1.insert(Mymap::value_type('b', 2));  
    c1.insert(Mymap::value_type('c', 3));  
  
    // find and show elements  
    std::cout << "c1.at('a') == " << c1.at('a') << std::endl;  
    std::cout << "c1.at('b') == " << c1.at('b') << std::endl;  
    std::cout << "c1.at('c') == " << c1.at('c') << std::endl;  
  
    return (0);  
}
```

unordered_map::begin

Designates the beginning of the controlled sequence or a bucket.

```
iterator begin();  
const_iterator begin() const;  
local_iterator begin(size_type nbucket);  
const_local_iterator begin(size_type nbucket) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

```
// std__unordered_map__unordered_map_begin.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    // inspect first two items " [c 3] [b 2]"
    Mymap::iterator it2 = c1.begin();
    std::cout << " [" << it2->first << ", " << it2->second << "]\n";
    ++it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]\n";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]\n";

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2]
[a, 1]
```

unordered_map::bucket

Gets the bucket number for a key value.

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

```
// std__unordered_map__unordered_map_bucket.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
               << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_map::bucket_count

Gets the number of buckets.

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

```

// std_unordered_map__unordered_map_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3][b, 2][a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_map::bucket_size

Gets the size of a bucket

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

```

// std__unordered_map__unordered_map_bucket_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_map::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

unordered_map::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();
// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

unordered_map::clear

Removes all elements.

```
void clear();
```

Remarks

The member function calls `unordered_map::erase` (`unordered_map::begin` (), `unordered_map::end` ()) .

Example

```
// std_unordered_map_unordered_map_clear.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" ;
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" ;
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_map::const_iterator

The type of a constant iterator for the controlled sequence.

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

```
// std_unordered_map_unordered_map_const_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";
    std::cout << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

```

// std__unordered_map__unordered_map_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]"<

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[a, 1]

```

unordered_map::const_pointer

The type of a constant pointer to an element.

```

typedef Alloc::const_pointer const_pointer;

```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_map_const_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::const_pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::const_reference

The type of a constant reference to an element.

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example


```

// std__unordered_map__unordered_map_const_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::const_reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::count

Finds the number of elements matching a specified key.

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by [unordered_map::equal_range](#) (`keyval`) .

Example

```

// std__unordered_map__unordered_map_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
count('A') == 0
count('b') == 1
count('C') == 0

```

unordered_map::difference_type

The type of a signed distance between two elements.

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

```

// std_unordered_map_unordered_map_difference_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // compute positive difference
    Mymap::difference_type diff = 0;
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Mymap::const_iterator it = c1.end();
         it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
end()-begin() == 3
begin()-end() == -3

```

unordered_map::emplace

Inserts an element constructed in place (no copy or move operations are performed) into an unordered_map.

```

template <class... Args>
pair<iterator, bool> emplace( Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered_map unless it already contains an element whose value is equivalently ordered.

Return Value

A `pair` whose **bool** component returns true if an insertion was made and false if the `unordered_map` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **bool** component of a pair `pr` returned by this member function, use `pr.second`.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [map::emplace](#).

unordered_map::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(const_iterator where, Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the <code>unordered_map</code> unless the <code>unordered_map</code> already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.
<i>where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No references are invalidated by this function.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see [map::emplace_hint](#).

unordered_map::empty

Tests whether no elements are present.

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

```
// std__unordered_map__unordered_map_empty.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << "\n";
    std::cout << "empty() == " << std::boolalpha << c1.empty() << "\n";
    std::cout << "\n";

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    std::cout << "size == " << c1.size() << "\n";
    std::cout << "empty() == " << std::boolalpha << c1.empty() << "\n";

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_map::end

Designates the end of the controlled sequence.

```
iterator end();
const_iterator end() const;
local_iterator end(size_type nbucket);
const_local_iterator end(size_type nbucket) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

unordered_map::equal_range

Finds range that matches a specified key.

```
std::pair<iterator, iterator> equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator> equal_range(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `x` such that `[x.first, x.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()` .

Example

```

// std_unordered_map__unordered_map_equal_range.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // display results of failed search
    std::pair<Mymap::iterator, Mymap::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
        << ", " << pair1.first->second << "]"
        std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
        << ", " << pair1.first->second << "]"
        std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
equal_range('x'):
equal_range('b'): [b, 2]

```

unordered_map::erase

Removes an element or a range of elements in a unordered_map from specified positions or removes elements that match a specified key.

```

iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);
size_type erase(const key_type& Key);

```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the `unordered_map`.

Remarks

For a code example, see [map::erase](#).

`unordered_map::find`

Finds an element that matches a specified key.

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns [unordered_map::equal_range](#)(`keyval`).first .

Example


```

// std__unordered_map__unordered_map_find.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Mymap::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << it->first << ", " << it->second << "]" << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
find('A') == false
find('b') == true: [b, 2]

```

unordered_map::get_allocator

Gets the stored allocator object.

```

Alloc get_allocator() const;

```

Remarks

The member function returns the stored allocator object.

Example

```
// std__unordered_map__unordered_map_get_allocator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

unordered_map::hash_function

Gets the stored hash function object.

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

```
// std__unordered_map__unordered_map_hash_function.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_map::hasher

The type of the hash function.

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

```
// std_unordered_map_unordered_map_hasher.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_map::insert

Inserts an element or a range of elements into an unordered_map.

```
// (1) single element
pair<iterator, bool> insert(    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(    ValTy&& Val);

// (3) single element with hint
iterator insert(    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(initializer_list<value_type>
    IList);
```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the <code>unordered_map</code> unless it already contains an element whose key is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion.
<i>ValTy</i>	Template parameter that specifies the argument type that the <code>unordered_map</code> can use to construct an element of <code>value_type</code> , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an <code>input iterator</code> that points to elements of a type that can be used to construct <code>value_type</code> objects.
<i>lList</i>	The <code>initializer_list</code> from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a `pair` whose `bool` component is true if an insertion was made, and false if the `unordered_map` already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the `bool` component is true, or to the existing element if the `bool` component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `unordered_map` or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

The `value_type` of a container is a typedef that belongs to the container, and for `map`, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into an `unordered_map` that

corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `m`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the `unordered_map`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [unordered_map::emplace](#) and [unordered_map::emplace_hint](#).

For a code example, see [map::insert](#).

unordered_map::iterator

The type of an iterator for the controlled sequence.

```
typedef T0 iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

Example

```
// std__unordered_map__unordered_map_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";
    std::cout << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::key_eq

Gets the stored comparison function object.

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```
// std_unordered_map_unordered_map_key_eq.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_map::key_equal

The type of the comparison function.

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

```
// std__unordered_map__unordered_map_key_equal.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_map::key_type

The type of an ordering key.

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

```

// std__unordered_map__unordered_map_key_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]

```

unordered_map::load_factor

Counts the average elements per bucket.

```
float load_factor() const;
```

Remarks

The member function returns `(float) unordered_map::size() / (float) unordered_map::bucket_count()`, the average number of elements per bucket.

Example


```

// std_unordered_map_unordered_map_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_map::local_iterator

The type of a bucket iterator.

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

```

// std_unordered_map_unordered_map_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[a, 1]

```

unordered_map::mapped_type

The type of a mapped value associated with each key.

```
typedef Ty mapped_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

```
// std__unordered_map__unordered_map_mapped_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

unordered_map::max_bucket_count

Gets the maximum number of buckets.

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

```

// std_unordered_map_unordered_map_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_map::max_load_factor

Gets or sets the maximum elements per bucket.

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example

```

// std_unordered_map_unordered_map_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" << "\n";

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_map::max_size

Gets the maximum size of the controlled sequence.

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

```

// std_unordered_map_unordered_map_max_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}

```

```
max_size() == 536870911
```

unordered_map::operator[]

Finds or inserts an element with the specified key.

```

Ty& operator[](const Key& keyval);

Ty& operator[](Key&& keyval);

```

Parameters

PARAMETER	DESCRIPTION
<i>Keyval</i>	The key value to find or insert.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map *m* using `m[Key] = DataValue`; where `DataValue` is the value of the `mapped_type` of the element with a key value of *Key*.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions [find](#) and [insert](#) can be used to determine whether an element with a specified key is already present before an insertion.

Example

```
// std_unordered_map_unordered_map_operator_sub.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
#include <string>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // try to find and fail
    std::cout << "c1['A'] == " << c1['A'] << std::endl;

    // try to find and succeed
    std::cout << "c1['a'] == " << c1['a'] << std::endl;

    // redisplay contents
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // insert by moving key
    std::unordered_map<string, int> c2;
    std::string str("abc");
    std::cout << "c2[std::move(str)] == " << c2[std::move(str)] << std::endl;
    std::cout << "c2[\"abc\"] == " << c2["abc"] << std::endl;

    return (0);
}
```



```

[c, 3] [b, 2] [a, 1]
c1['A'] == 0
c1['a'] == 1
[c, 3] [b, 2] [A, 0] [a, 1]
c2[move(str)] == 0
c2["abc"] == 1

```

Remarks

The member function determines the iterator `where` as the return value of `unordered_map::insert` (`unordered_map::value_type` (`keyval`, `Ty()`)). (It inserts an element with the specified key if no such element exists.) It then returns a reference to `(*where).second`.

unordered_map::operator=

Replaces the elements of this `unordered_map` using the elements from another `unordered_map`.

```

unordered_map& operator=(const unordered_map& right);

unordered_map& operator=(unordered_map&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>unordered_map</code> that the operator function assigns content from.

Remarks

The first version copies all of the elements from *right* to this `unordered_map`.

The second version moves all of the elements from *right* to this `unordered_map`.

Any elements that are in this `unordered_map` before `operator =` executes are discarded.

Example

```

// unordered_map_operator_as.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

int main( )
{
    using namespace std;
    unordered_map<int, int> v1, v2, v3;
    unordered_map<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}

```

unordered_map::pointer

The type of a pointer to an element.

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_map_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::reference

The type of a reference to an element.

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_map_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::rehash

Rebuilds the hash table.

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

```

// std_unordered_map__unordered_map_rehash.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1

```

unordered_map::size

Counts the number of elements.

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

```
// std__unordered_map__unordered_map_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_map::size_type

The type of an unsigned distance between two elements.

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled

sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

```
// std__unordered_map__unordered_map_size_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;
    Mymap::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

```
size == 0
```

unordered_map::swap

Swaps the contents of two containers.

```
void swap(unordered_map& right);
```

Parameters

right

The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `unordered_map::get_allocator()` `() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

```

// std__unordered_map__unordered_map_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

    // display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
    std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]

```

unordered_map::unordered_map

Constructs a container object.


```

unordered_map(const unordered_map& Right);

explicit unordered_map(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Allocator());

unordered_map(unordered_map&& Right);
unordered_map(initializer_list<Type> IList);
unordered_map(initializer_list<Type> IList, size_type Bucket_count);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    KeyEqual& equal);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    KeyEqual& Equal
    const Allocator& Al);

template <class InIt>
unordered_map(
    InputIterator First,
    InputIterator Last,
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The allocator object to store.
<i>Comp</i>	The comparison function object to store.
<i>Hash</i>	The hash function object to store.
<i>Bucket_count</i>	The minimum number of buckets.
<i>Right</i>	The container to copy.
<i>First</i>	
<i>Last</i>	
<i>IList</i>	The initializer_list that contains the elements to be copied.

Remarks

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`. The fourth constructor specifies a copy of the sequence by moving `right`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

the minimum number of buckets is the argument *Bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `N0`.

the hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Pred()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

Example

```
// std__unordered_map__unordered_map_construct.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
#include <initializer_list>

using namespace std;

using Mymap = unordered_map<char, int>;

int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]"<< endl;
    }

    Mymap c2(8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    // display contents " [f 6] [e 5] [d 4]"
    for (const auto& c : c2) {
        cout << " [" << c.first << ", " << c.second << "]"<< endl;
    }

    Mymap c3(c1.begin(),
        c1.end(),
        8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c3) {
```

```

        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;

    Mymap c4(move(c3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c4) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;
    cout << endl;

    // Construct with an initializer_list
    unordered_map<int, char> c5({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } });
    for (const auto& c : c5) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;

    // Initializer_list plus size
    unordered_map<int, char> c6({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } }, 4);
    for (const auto& c : c6) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;
    cout << endl;

    // Initializer_list plus size and hash
    unordered_map<int, char, hash<char>> c7(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>()
    );

    for (const auto& c : c7) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;

    // Initializer_list plus size, hash, and key_equal
    unordered_map<int, char, hash<char>, equal_to<char>> c8(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>()
    );

    for (const auto& c : c8) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;

    // Initializer_list plus size, hash, key_equal, and allocator
    unordered_map<int, char, hash<char>, equal_to<char>>, allocator<pair<const char, int>> c9(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int>>()
    );

    for (const auto& c : c9) {
        cout << " [" << c.first << ", " << c.second << "]\n";
    }
    cout << endl;
}

```

```
[a, 1] [b, 2] [c, 3]
[d, 4] [e, 5] [f, 6]
[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]

[5, g] [6, h] [7, i] [8, j]
[a, 1] [b, 2] [c, 3]

[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]
```

unordered_map::value_type

The type of an element.

```
typedef std::pair<const Key, Ty> value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

```
// std__unordered_map__unordered_map_value_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

See also

[<unordered_map>](#)

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

unordered_multimap Class

11/9/2018 • 42 minutes to read • [Edit Online](#)

The template class describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element stores two objects, a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

```
template <class Key,  
          class Ty,  
          class Hash = std::hash<Key>,  
          class Pred = std::equal_to<Key>,  
          class Alloc = std::allocator<Key>>  
class unordered_multimap;
```

Parameters

PARAMETER	DESCRIPTION
<i>Key</i>	The key type.
<i>Ty</i>	The mapped type.
<i>Hash</i>	The hash function object type.
<i>Pred</i>	The equality comparison function object type.
<i>Alloc</i>	The allocator class.

Members

TYPE DEFINITION	DESCRIPTION
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.

TYPE DEFINITION	DESCRIPTION
<code>const_pointer</code>	The type of a constant pointer to an element.
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>mapped_type</code>	The type of a mapped value associated with each key.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.
MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Designates the beginning of the controlled sequence.
<code>bucket</code>	Gets the bucket number for a key value.
<code>bucket_count</code>	Gets the number of buckets.
<code>bucket_size</code>	Gets the size of a bucket.
<code>cbegin</code>	Designates the beginning of the controlled sequence.
<code>cend</code>	Designates the end of the controlled sequence.
<code>clear</code>	Removes all elements.
<code>count</code>	Finds the number of elements matching a specified key.
<code>emplace</code>	Adds an element constructed in place.
<code>emplace_hint</code>	Adds an element constructed in place, with hint.

MEMBER FUNCTION	DESCRIPTION
<code>empty</code>	Tests whether no elements are present.
<code>end</code>	Designates the end of the controlled sequence.
<code>equal_range</code>	Finds range that matches a specified key.
<code>erase</code>	Removes elements at specified positions.
<code>find</code>	Finds an element that matches a specified key.
<code>get_allocator</code>	Gets the stored allocator object.
<code>hash_function</code>	Gets the stored hash function object.
<code>insert</code>	Adds elements.
<code>key_eq</code>	Gets the stored comparison function object.
<code>load_factor</code>	Counts the average elements per bucket.
<code>max_bucket_count</code>	Gets the maximum number of buckets.
<code>max_load_factor</code>	Gets or sets the maximum elements per bucket.
<code>max_size</code>	Gets the maximum size of the controlled sequence.
<code>rehash</code>	Rebuilds the hash table.
<code>size</code>	Counts the number of elements.
<code>swap</code>	Swaps the contents of two containers.
<code>unordered_multimap</code>	Constructs a container object.

OPERATOR	DESCRIPTION
<code>unordered_multimap::operator=</code>	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `unordered_multimap::key_equal` and a hash function object of type `unordered_multimap::hasher`. You access the first stored object by calling the member function `unordered_multimap::key_eq()`; and you access the second stored object by calling the member function `unordered_multimap::hash_function()`. Specifically, for all values `x` and `y` of type `Key`, the call `key_eq()(x, y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_map Class`, an object of template class `unordered_multimap` does not ensure that `key_eq()(x, y)` is always false for any two elements of the controlled sequence. (Keys need not be unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `unordered_multimap::load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type `unordered_multimap::allocator_type`. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: `<unordered_map>`

Namespace: `std`

`unordered_multimap::allocator_type`

The type of an allocator for managing storage.

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

```
// std__unordered_map__unordered_multimap_allocator_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

`unordered_multimap::begin`

Designates the beginning of the controlled sequence or a bucket.

```

iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;

```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

```

// std__unordered_map__unordered_multimap_begin.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" << "\n";

    // inspect first two items " [c 3] [b 2]"
    Mymap::iterator it2 = c1.begin();
    std::cout << " [" << it2->first << ", " << it2->second << "]" << "\n";
    ++it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]" << "\n";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]" << "\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2]
[a, 1]

```

unordered_multimap::bucket

Gets the bucket number for a key value.

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

```
// std__unordered_map__unordered_multimap_bucket.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]\n";
    std::cout << std::endl;

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multimap::bucket_count

Gets the number of buckets.

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

```
// std__unordered_map__unordered_multimap_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_multimap::bucket_size

Gets the size of a bucket

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

```

// std__unordered_map__unordered_multimap_bucket_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << it->first << ", " << it->second << "]"<< "\n";

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multimap::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

unordered_multimap::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

unordered_multimap::clear

Removes all elements.

```
void clear();
```

Remarks

The member function calls `unordered_multimap::erase (unordered_multimap::begin (), unordered_multimap::end ())`.

Example

```
// std_unordered_map_unordered_multimap_clear.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::const_iterator

The type of a constant iterator for the controlled sequence.

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

```
// std__unordered_map__unordered_multimap_const_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";
    std::cout << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

```
// std__unordered_map__unordered_multimap_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]"

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

unordered_multimap::const_pointer

The type of a constant pointer to an element.

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_multimap_const_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::const_pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::const_reference

The type of a constant reference to an element.

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

```

// std_unordered_map__unordered_multimap_const_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::const_reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::count

Finds the number of elements matching a specified key.

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by [unordered_multimap::equal_range](#) (`keyval`).

Example

```

// std_unordered_map_unordered_multimap_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
count('A') == 0
count('b') == 1
count('C') == 0

```

unordered_multimap::difference_type

The type of a signed distance between two elements.

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

```

// std_unordered_map_unordered_multimap_difference_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    // compute positive difference
    Mymap::difference_type diff = 0;
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Mymap::const_iterator it = c1.end();
         it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
end()-begin() == 3
begin()-end() == -3

```

unordered_multimap::emplace

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```

template <class... Args>
iterator emplace(Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered_multimap.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [multimap::emplace](#).

unordered_multimap::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered.
<i>where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

The [value_type](#) of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see [map::emplace_hint](#).

unordered_multimap::empty

Tests whether no elements are present.

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

```
// std__unordered_map__unordered_multimap_empty.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::end

Designates the end of the controlled sequence.

```

iterator end();

const_iterator end() const;

local_iterator end(size_type nbucket);

const_local_iterator end(size_type nbucket) const;

```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

```

// std_unordered_map_unordered_multimap_end.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // inspect last two items " [a 1] [b 2]"
    Mymap::iterator it2 = c1.end();
    --it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]"<< "\n";
    --it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]"<< "\n";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << " [" << lit->first << ", " << lit->second << "]"<< "\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[a, 1] [b, 2]
[a, 1]

```


unordered_multimap::equal_range

Finds range that matches a specified key.

```
std::pair<iterator, iterator>
    equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
    equal_range(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `x` such that `[x.first, x.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

```
// std__unordered_map__unordered_multimap_equal_range.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // display results of failed search
    std::pair<Mymap::iterator, Mymap::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
            << ", " << pair1.first->second << "]"
        std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
            << ", " << pair1.first->second << "]"
        std::cout << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
equal_range('x'):
equal_range('b'): [b, 2]
```

unordered_multimap::erase

Removes an element or a range of elements in a `unordered_multimap` from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the `unordered_multimap`.

Remarks

For a code example, see [map::erase](#).

unordered_multimap::find

Finds an element that matches a specified key.

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns `unordered_multimap::equal_range` `(keyval).first` .

Example

```
// std__unordered_map__unordered_multimap_find.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    // try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Mymap::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << it->first << ", " << it->second << "]" << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
find('A') == false
find('b') == true: [b, 2]
```

unordered_multimap::get_allocator

Gets the stored allocator object.

```
Alloc get_allocator() const;
```

Remarks

The member function returns the stored allocator object.

Example

```
// std__unordered_map__unordered_multimap_get_allocator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

unordered_multimap::hash_function

Gets the stored hash function object.

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

```
// std__unordered_map__unordered_multimap_hash_function.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multimap::hasher

The type of the hash function.

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

```
// std__unordered_map__unordered_multimap_hasher.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multimap::insert

Inserts an element or a range of elements into an unordered_multimap.

```

// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);

```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the <code>unordered_multimap</code> .
<i>Where</i>	The place to start searching for the correct point of insertion.
<i>ValTy</i>	Template parameter that specifies the argument type that the <code>unordered_multimap</code> can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the `unordered_multimap`.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `unordered_multimap`.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The [value_type](#) of a container is a typedef that belongs to the container, and for `map`,

`map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into an `unordered_multimap` that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *Last* does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the `unordered_multimap`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [unordered_multimap::emplace](#) and [unordered_multimap::emplace_hint](#).

For a code example, see [multimap::insert](#).

unordered_multimap::iterator

The type of an iterator for the controlled sequence.

```
typedef T0 iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

Example

```

// std__unordered_map__unordered_multimap_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::key_eq

Gets the stored comparison function object.

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```

// std__unordered_map__unordered_multimap_key_eq.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}

```

```

cmpfn('a', 'a') == true
cmpfn('a', 'b') == false

```


unordered_multimap::key_equal

The type of the comparison function.

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

```
// std__unordered_map__unordered_multimap_key_equal.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multimap::key_type

The type of an ordering key.

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

```

// std_unordered_map_unordered_multimap_key_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<< "\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]

```

unordered_multimap::load_factor

Counts the average elements per bucket.

```
float load_factor() const;
```

Remarks

The member function returns `(float) unordered_multimap::size() / (float) unordered_multimap::bucket_count()`, the average number of elements per bucket.

Example

```

// std_unordered_map_unordered_multimap_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

unordered_multimap::local_iterator

The type of a bucket iterator.

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

```
// std__unordered_map__unordered_multimap_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]"
    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

unordered_multimap::mapped_type

The type of a mapped value associated with each key.

```
typedef Ty mapped_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

```

// std_unordered_map_unordered_multimap_mapped_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]

```

unordered_multimap::max_bucket_count

Gets the maximum number of buckets.

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

```

// std_unordered_map_unordered_multimap_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multimap::max_load_factor

Gets or sets the maximum elements per bucket.

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example

```

// std_unordered_map_unordered_multimap_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```



```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_multimap::max_size

Gets the maximum size of the controlled sequence.

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

```

// std__unordered_map__unordered_multimap_max_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}

```

```
max_size() == 536870911
```

unordered_multimap::operator=

Copies a hash table.

```

unordered_multimap& operator=(const unordered_multimap& right);

unordered_multimap& operator=(unordered_multimap&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The unordered_multimap being copied into the unordered_multimap.

Remarks

After erasing any existing elements in a unordered_multimap, `operator=` either copies or moves the contents of *right* into the unordered_multimap.

Example

```
// unordered_multimap_operator_as.cpp
// compile with: /EHsc
#include <unordered_multimap>
#include <iostream>

int main( )
{
    using namespace std;
    unordered_multimap<int, int> v1, v2, v3;
    unordered_multimap<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << iter->second << " ";
    cout << endl;
}
```

unordered_multimap::pointer

The type of a pointer to an element.

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_multimap_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::reference

The type of a reference to an element.

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

```

// std__unordered_map__unordered_multimap_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Mymap::reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]"<< "\n";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::rehash

Rebuilds the hash table.

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

```

// std_unordered_map_unordered_multimap_rehash.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" << "\n";

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << "\n";
    std::cout << "load_factor() == " << c1.load_factor() << "\n";
    std::cout << "max_load_factor() == " << c1.max_load_factor() << "\n";

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << "\n";
    std::cout << "load_factor() == " << c1.load_factor() << "\n";
    std::cout << "max_load_factor() == " << c1.max_load_factor() << "\n";

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << "\n";
    std::cout << "load_factor() == " << c1.load_factor() << "\n";
    std::cout << "max_load_factor() == " << c1.max_load_factor() << "\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1

```

unordered_multimap::size

Counts the number of elements.

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

```
// std__unordered_map__unordered_multimap_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"
        std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::size_type

The type of an unsigned distance between two elements.

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled

sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

```
// std__unordered_map__unordered_multimap_size_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;
    Mymap::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

```
size == 0
```

unordered_multimap::swap

Swaps the contents of two containers.

```
void swap(unordered_multimap& right);
```

Parameters

right

The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `unordered_multimap::get_allocator()` `() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

```

// std__unordered_map__unordered_multimap_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

    // display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]"<
        std::cout << std::endl;

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]

```

unordered_multimap::unordered_multimap

Constructs a container object.


```

unordered_multimap(
    const unordered_multimap& Right);

explicit unordered_multimap(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Pred(),
    const Allocator& Al = Alloc());

unordered_multimap(
    unordered_multimap&& Right);

unordered_multimap(
    initializer_list<Type> IList);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key,
    const Allocator& Al);

template <class InputIterator>
unordered_multimap(
    InputIterator first,
    InputIterator last,
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Pred(),
    const Allocator& Al = Alloc());

```

Parameters

PARAMETER	DESCRIPTION
<i>InputIterator</i>	The iterator type.
<i>Al</i>	The allocator object to store.
<i>Comp</i>	The comparison function object to store.
<i>Hash</i>	The hash function object to store.
<i>Bucket_count</i>	The minimum number of buckets.
<i>Right</i>	The container to copy.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list from which to copy the elements.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor specifies an empty controlled sequence. The third constructor. specifies a copy of the sequence by moving *Right*. The fourth, fifth, sixth, seventh, and eighth constructors use an initializer_list for the members. The ninth constructor inserts the sequence of element values `[First, Last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *Bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `N0`.

The hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Pred()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

Example

```
// std__unordered_map__unordered_multimap_construct.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

using namespace std;

using Mymap = unordered_multimap<char, int> ;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]" ;
    }
    cout << endl;

    Mymap c2(8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    // display contents " [f 6] [e 5] [d 4]"
    for (const auto& c : c2) {
        cout << " [" << c.first << ", " << c.second << "]" ;
    }
    cout << endl;

    Mymap c3(c1.begin(),
        c1.end(),
        8.
```

```

~,
hash<char>(),
equal_to<char>(),
allocator<pair<const char, int> >());

// display contents " [c 3] [b 2] [a 1]"
for (const auto& c : c3) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

Mymap c4(move(c3));

// display contents " [c 3] [b 2] [a 1]"
for (const auto& c : c4) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

// Construct with an initializer_list
unordered_multimap<int, char> c5({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } });
for (const auto& c : c5) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

// Initializer_list plus size
unordered_multimap<int, char> c6({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } }, 4);
for (const auto& c : c1) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

// Initializer_list plus size and hash
unordered_multimap<int, char, hash<char>> c7(
    { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
    4,
    hash<char>()
);

for (const auto& c : c1) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

// Initializer_list plus size, hash, and key_equal
unordered_multimap<int, char, hash<char>, equal_to<char>> c8(
    { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
    4,
    hash<char>(),
    equal_to<char>()
);

for (const auto& c : c1) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

// Initializer_list plus size, hash, key_equal, and allocator
unordered_multimap<int, char, hash<char>, equal_to<char>,
    allocator<pair<const char, int> >()
);

for (const auto& c : c1) {
    cout << " [" << c.first << ", " << c.second << "]"<endl;
}

```

```

    }
    cout << endl;
}

```

```

[a, 1] [b, 2] [c, 3] [d, 4] [e, 5] [f, 6] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3] [5, g] [6, h]
[7, i] [8, j] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3]
[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2] [a, 1]

```

unordered_multimap::value_type

The type of an element.

```

typedef std::pair<const Key, Ty> value_type;

```

Remarks

The type describes an element of the controlled sequence.

Example

```

// std_unordered_map_unordered_multimap_value_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" << "\n";

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]" << "\n";

    return (0);
}

```

```

[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]

```

See also

[<unordered_map>](#)

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<unordered_set>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template classes [unordered_multiset](#) and [unordered_set](#) and their supporting templates.

Syntax

```
#include <unordered_set>
```

Classes

CLASS	DESCRIPTION
unordered_multiset Class	Stores hash table of keys.
unordered_set Class	Stores hash table of keys.

Functions

FUNCTION	DESCRIPTION
operator!=	Tests if the unordered_multiset object on the left side of the operator is not equal to the unordered_multiset object on the right side.
operator==	Tests if the unordered_multiset object on the left side of the operator is equal to the unordered_multiset object on the right side.
swap	Swaps two multisets.
operator!=	Tests if the unordered_set object on the left side of the operator is not equal to the unordered_set object on the right side.
operator==	Tests if the unordered_set object on the left side of the operator is equal to the unordered_set object on the right side.
swap	Swaps two sets.

See also

[unordered_map Class](#)

[unordered_multimap Class](#)

<unordered_set> functions

11/9/2018 • 2 minutes to read • [Edit Online](#)

`swap (set)`

`swap (unordered_multiset)`

swap (unordered_set)

Swaps the contents of two containers.

```
template <class Key, class Hash, class Pred, class Alloc>
void swap(
    unordered_set <Key, Hash, Pred, Alloc>& left,
    unordered_set <Key, Hash, Pred, Alloc>& right);
```

Parameters

Key

The key type.

Hash

The hash function object type.

Pred

The equality comparison function object type.

Alloc

The allocator class.

left

The first container to swap.

right

The second container to swap.

Remarks

The template function executes `left.unordered_set::swap (right)`.

Example

```

#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << " ]";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents " [f] [e] [d]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << " ]";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << " ]";
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
[f] [e] [d]
[c] [b] [a]

```

swap (unordered_multiset)

Swaps the contents of two containers.

```

template <class Key, class Hash, class Pred, class Alloc>
void swap(
    unordered_multiset <Key, Hash, Pred, Alloc>& left,
    unordered_multiset <Key, Hash, Pred, Alloc>& right);

```

Parameters

Key

The key type.

Hash

The hash function object type.

Pred

The equality comparison function object type.

Alloc

The allocator class.

left

The first container to swap.

right

The second container to swap.

Remarks

The template function executes `left. unordered_multiset::swap (right)`.

Example

```
// std__unordered_set__u_ms_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << "]\n";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents " [f] [e] [d]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << "]\n";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << " [" << *it << "]\n";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]  
[f] [e] [d]  
[c] [b] [a]
```

See also

[`<unordered_set>`](#)

<unordered_set> operators

11/9/2018 • 4 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator==</code>	<code>operator!=</code>	<code>operator==</code>

operator!=

Tests whether the `unordered_set` object on the left side of the operator is not equal to the `unordered_set` object on the right side.

```
bool operator!=(const unordered_set <Key, Hash, Pred, Allocator>& left, const unordered_set <Key, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_set`.

right

An object of type `unordered_set`.

Return Value

true if the `unordered_sets` are not equal; **false** if they are equal.

Remarks

The comparison between `unordered_set` objects is not affected by the arbitrary order in which they store their elements. Two `unordered_sets` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```

// unordered_set_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_set<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

    cout << boolalpha;
    cout << "c1 != c2: " << (c1 != c2) << endl;
    cout << "c1 != c3: " << (c1 != c3) << endl;
    cout << "c2 != c3: " << (c2 != c3) << endl;

    return (0);
}

```

Output:

```
c1 != c2: true
```

```
c1 != c3: false
```

```
c2 != c3: true
```

operator==

Tests whether the [unordered_set](#) object on the left side of the operator is equal to the `unordered_set` object on the right side.

```
bool operator==(const unordered_set <Key, Hash, Pred, Allocator>& left, const unordered_set <Key, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_set`.

right

An object of type `unordered_set`.

Return Value

true if the `unordered_sets` are equal; **false** if they are not equal.

Remarks

The comparison between `unordered_set` objects is not affected by the arbitrary order in which they store their

elements. Two `unordered_sets` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```
// unordered_set_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_set<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

    cout << boolalpha;
    cout << "c1 == c2: " << (c1 == c2) << endl;
    cout << "c1 == c3: " << (c1 == c3) << endl;
    cout << "c2 == c3: " << (c2 == c3) << endl;

    return (0);
}
```

Output:

```
c1 == c2: false
```

```
c1 == c3: true
```

```
c2 == c3: false
```

operator!=

Tests whether the [unordered_multiset](#) object on the left side of the operator is not equal to the `unordered_multiset` object on the right side.

```
bool operator!=(const unordered_multiset <Key, Hash, Pred, Allocator>& left, const unordered_multiset <Key, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_multiset`.

right

An object of type `unordered_multiset`.

Return Value

true if the unordered_multisets are not equal; **false** if they are equal.

Remarks

The comparison between unordered_multiset objects is not affected by the arbitrary order in which they store their elements. Two unordered_multisets are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```
// unordered_multiset_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_multiset<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
    c1.insert('c');

    c2.insert('c');
    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

    cout << boolalpha;
    cout << "c1 != c2: " << (c1 != c2) << endl;
    cout << "c1 != c3: " << (c1 != c3) << endl;
    cout << "c2 != c3: " << (c2 != c3) << endl;

    return (0);
}
```

Output:

```
c1 != c2: true
```

```
c1 != c3: false
```

```
c2 != c3: true
```

operator==

Tests whether the [unordered_multiset](#) object on the left side of the operator is equal to the unordered_multiset object on the right side.

```
bool operator==(const unordered_multiset <Key, Hash, Pred, Allocator>& left, const unordered_multiset <Key, Hash, Pred, Allocator>& right);
```

Parameters

left

An object of type `unordered_multiset`.

right

An object of type `unordered_multiset`.

Return Value

true if the `unordered_multisets` are equal; **false** if they are not equal.

Remarks

The comparison between `unordered_multiset` objects is not affected by the arbitrary order in which they store their elements. Two `unordered_multisets` are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

```
// unordered_multiset_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_multiset<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
    c1.insert('c');

    c2.insert('c');
    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

    cout << boolalpha;
    cout << "c1 == c2: " << (c1 == c2) << endl;
    cout << "c1 == c3: " << (c1 == c3) << endl;
    cout << "c2 == c3: " << (c2 == c3) << endl;

    return (0);
}
```

Output:

```
c1 == c2: false
```

```
c1 == c3: true
```

```
c2 == c3: false
```

See also

[<unordered_set>](#)

unordered_set Class

10/31/2018 • 36 minutes to read • [Edit Online](#)

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

```
template <
    class Key,
    class Hash = std::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key>>
class unordered_set;
```

Parameters

PARAMETER	DESCRIPTION
<i>Key</i>	The key type.
<i>Hash</i>	The hash function object type.
<i>Pred</i>	The equality comparison function object type.
<i>Alloc</i>	The allocator class.

Members

TYPE DEFINITION	DESCRIPTION
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.

TYPE DEFINITION	DESCRIPTION
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.
MEMBER FUNCTION	DESCRIPTION
<code>begin</code>	Designates the beginning of the controlled sequence.
<code>bucket</code>	Gets the bucket number for a key value.
<code>bucket_count</code>	Gets the number of buckets.
<code>bucket_size</code>	Gets the size of a bucket.
<code>cbegin</code>	Designates the beginning of the controlled sequence.
<code>cend</code>	Designates the end of the controlled sequence.
<code>clear</code>	Removes all elements.
<code>count</code>	Finds the number of elements matching a specified key.
<code>emplace</code>	Adds an element constructed in place.
<code>emplace_hint</code>	Adds an element constructed in place, with hint.
<code>empty</code>	Tests whether no elements are present.
<code>end</code>	Designates the end of the controlled sequence.

MEMBER FUNCTION	DESCRIPTION
<code>equal_range</code>	Finds range that matches a specified key.
<code>erase</code>	Removes elements at specified positions.
<code>find</code>	Finds an element that matches a specified key.
<code>get_allocator</code>	Gets the stored allocator object.
<code>hash_function</code>	Gets the stored hash function object.
<code>insert</code>	Adds elements.
<code>key_eq</code>	Gets the stored comparison function object.
<code>load_factor</code>	Counts the average elements per bucket.
<code>max_bucket_count</code>	Gets the maximum number of buckets.
<code>max_load_factor</code>	Gets or sets the maximum elements per bucket.
<code>max_size</code>	Gets the maximum size of the controlled sequence.
<code>rehash</code>	Rebuilds the hash table.
<code>size</code>	Counts the number of elements.
<code>swap</code>	Swaps the contents of two containers.
<code>unordered_set</code>	Constructs a container object.

OPERATORS	DESCRIPTION
<code>unordered_set::operator=</code>	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `unordered_set::key_equal` and a hash function object of type `unordered_set::hasher`. You access the first stored object by calling the member function `unordered_set::key_eq()`; and you access the second stored object by calling the member function `unordered_set::hash_function()`. Specifically, for all values `x` and `y` of type `key`, the call `key_eq()(x, y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_multiset` Class, an object of template class `unordered_set` ensures that `key_eq()(x, y)` is always false for any two elements of the controlled sequence. (Keys are unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `unordered_set::load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type `unordered_set::allocator_type`. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: `<unordered_set>`

Namespace: `std`

`unordered_set::allocator_type`

The type of an allocator for managing storage.

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

```
// std__unordered_set__unordered_set_allocator_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
    << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

`unordered_set::begin`

Designates the beginning of the controlled sequence or a bucket.

```
iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

```

// unordered_set_begin.cpp
// compile using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>

using namespace std;

typedef unordered_set<char> MySet;

int main()
{
    MySet c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents using range-based for
    for (auto it : c1) {
        cout << "[" << it << "]" ";
    }

    cout << endl;

    // display contents using explicit for
    for (MySet::const_iterator it = c1.begin(); it != c1.end(); ++it) {
        cout << "[" << *it << "]" ";
    }

    cout << std::endl;

    // display first two items
    MySet::iterator it2 = c1.begin();
    cout << "[" << *it2 << "]" ";
    ++it2;
    cout << "[" << *it2 << "]" ";
    cout << endl;

    // display bucket containing 'a'
    MySet::const_local_iterator lit = c1.begin(c1.bucket('a'));
    cout << "[" << *lit << "]" ";

    return (0);
}

```

```

[a] [b] [c]
[a] [b] [c]
[a] [b]
[a]

```

unordered_set::bucket

Gets the bucket number for a key value.

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

```
// std__unordered_set__unordered_set_bucket.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
    << std::endl;

    return (0);
}
```

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_set::bucket_count

Gets the number of buckets.

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

```

// std_unordered_set_unordered_set_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_set::bucket_size

Gets the size of a bucket

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

```

// std__unordered_set__unordered_set_bucket_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
    << std::endl;

    return (0);
}

```



```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_set::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 isContainer<T>::iterator
auto i2 = Container.cbegin();

// i2 isContainer<T>::const_iterator
```

unordered_set::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 isContainer<T>::iterator
auto i2 = Container.cend();

// i2 isContainer<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

unordered_set::clear

Removes all elements.

```
void clear();
```

Remarks

The member function calls `unordered_set::erase` (`unordered_set::begin` (), `unordered_set::end` ()).

Example

```
// std__unordered_set__unordered_set_clear.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c] [b] [a]
size == 0
empty() == true
[e] [d]
size == 2
empty() == false
```

unordered_set::const_iterator

The type of a constant iterator for the controlled sequence.

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

```
// std__unordered_set__unordered_set_const_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_set::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

```

// std__unordered_set__unordered_set_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "]" ";

    return (0);
}

```

```

[c] [b] [a]
[a]

```

unordered_set::const_pointer

The type of a constant pointer to an element.

```

typedef Alloc::const_pointer const_pointer;

```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

```
// std__unordered_set__unordered_set_const_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::const_pointer p = &*it;
        std::cout << "[" << *p << " ] ";
    }
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_set::const_reference

The type of a constant reference to an element.

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

```
// std__unordered_set__unordered_set_const_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::const_reference ref = *it;
        std::cout << "[" << ref << "]" ";
    }
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_set::count

Finds the number of elements matching a specified key.

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by [unordered_set::equal_range](#) (`keyval`) .

Example

```

// std__unordered_set__unordered_set_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}

```

```

[c] [b] [a]
count('A') == 0
count('b') == 1
count('C') == 0

```

unordered_set::difference_type

The type of a signed distance between two elements.

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

```
// std__unordered_set__unordered_set_difference_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // compute positive difference
    Myset::difference_type diff = 0;
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Myset::const_iterator it = c1.end(); it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}
```

```
[c] [b] [a]
end()-begin() == 3
begin()-end() == -3
```

unordered_set::emplace

Inserts an element constructed in place (no copy or move operations are performed).

```
template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the <code>unordered_set</code> unless it already contains an element whose value is equivalently ordered.

Return Value

A `pair` whose **bool** component returns true if an insertion was made and false if the `unordered_set` already contained an element whose key had an equivalent value in the ordering,

and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **bool** component of a pair `pr` returned by this member function, use `pr.second`.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [set::emplace](#).

unordered_set::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iteratorwhere,
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered_set unless the unordered_set already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.
<i>where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [set::emplace_hint](#).

unordered_set::empty

Tests whether no elements are present.

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

```
// std__unordered_set__unordered_set_empty.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c] [b] [a]
size == 0
empty() == true
[e] [d]
size == 2
empty() == false
```

unordered_set::end

Designates the end of the controlled sequence.

```

iterator end();

const_iterator end() const;

local_iterator end(size_type nbucket);

const_local_iterator end(size_type nbucket) const;

```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

```

// std__unordered_set__unordered_set_end.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    // inspect last two items "[a] [b] "
    Myset::iterator it2 = c1.end();
    --it2;
    std::cout << "[" << *it2 << " ] ";
    --it2;
    std::cout << "[" << *it2 << " ] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << "[" << *lit << " ] ";

    return (0);
}

```

```

[c] [b] [a]
[a] [b]
[a]

```

unordered_set::equal_range

Finds range that matches a specified key.

```
std::pair<iterator, iterator>
equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
equal_range(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `x` such that `[x.first, x.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

```
// std__unordered_set__unordered_set_equal_range.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    // display results of failed search
    std::pair<Myset::iterator, Myset::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << " ] ";
    std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << " ] ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
equal_range('x'):
equal_range('b'): [b]
```

unordered_set::erase

Removes an element or a range of elements in a `unordered_set` from specified positions or removes elements that match a specified key.

```
iterator erase(const_iterator Where);

iterator erase(const_iterator First, const_iterator Last);

size_type erase(const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the `unordered_set` if no such element exists.

For the third member function, returns the number of elements that have been removed from the `unordered_set`.

Remarks

For a code example, see [set::erase](#).

unordered_set::find

Finds an element that matches a specified key.

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns `unordered_set::equal_range` `(keyval).first`.

Example

```

// std__unordered_set__unordered_set_find.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // try to find and fail
    std::cout << "find('A') == "
    << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Myset::iterator it = c1.find('b');
    std::cout << "find('b') == "
    << std::boolalpha << (it != c1.end())
    << ": [" << *it << "]" " << std::endl;

    return (0);
}

```

```

[c] [b] [a]
find('A') == false
find('b') == true: [b]

```

unordered_set::get_allocator

Gets the stored allocator object.

```

Alloc get_allocator() const;

```

Remarks

The member function returns the stored allocator object.

Example

```
// std__unordered_set__unordered_set_get_allocator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
    << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

unordered_set::hash_function

Gets the stored hash function object.

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

```
// std__unordered_set__unordered_set_hash_function.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_set::hasher

The type of the hash function.

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

```
// std__unordered_set__unordered_set_hasher.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_set::insert

Inserts an element or a range of elements into an unordered_set.

```
// (1) single element
pair<iterator, bool> insert(const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool> insert(ValTy&& Val);

// (3) single element with hint
iterator insert(const_iterator Where, const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(const_iterator Where, ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(InputIterator First, InputIterator Last);

// (6) initializer list
void insert(initializer_list<value_type> IList);
```

Parameters

PARAMETER	DESCRIPTION
-----------	-------------

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the <code>unordered_set</code> unless it already contains an element whose key is equivalently ordered.
<i>Where</i>	The place to start searching for the correct point of insertion.
<i>ValTy</i>	Template parameter that specifies the argument type that the <code>unordered_set</code> can use to construct an element of <code>value_type</code> , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an <code>input iterator</code> that points to elements of a type that can be used to construct <code>value_type</code> objects.
<i>lList</i>	The <code>initializer_list</code> from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a `pair` whose **bool** component is true if an insertion was made, and false if the `unordered_set` already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is true, or to the existing element if the **bool** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `unordered_set` or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the **bool** component, use `pr.second`. For an example, see the sample code later in this article.

The `value_type` of a container is a typedef that belongs to the container, and, for set,

```
unordered_set<V>::value_type is type const V.
```

The range member function (5) inserts the sequence of element values into an `unordered_set` that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *Last* does not get inserted. The container member function `end()` refers to the position just after the last

element in the container—for example, the statement `s.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `s`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an [initializer_list](#) to copy elements into the `unordered_set`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see [set::emplace](#) and [set::emplace_hint](#).

For a code example, see [set::insert](#).

unordered_set::iterator

A type that provides a constant [forward iterator](#) that can read elements in an `unordered_set`.

```
typedef implementation-defined iterator;
```

Example

See the example for [begin](#) for an example of how to declare and use an **iterator**.

unordered_set::key_eq

Gets the stored comparison function object.

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```
// std__unordered_set__unordered_set_key_eq.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
    << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
    << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_set::key_equal

The type of the comparison function.

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

```
// std__unordered_set__unordered_set_key_equal.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
    << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
    << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_set::key_type

The type of an ordering key.

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

```

// std__unordered_set__unordered_set_key_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
[d] [c] [b] [a]

```

unordered_set::load_factor

Counts the average elements per bucket.

```
float load_factor() const;
```

Remarks

The member function returns `(float) unordered_set::size () / (float) unordered_set::bucket_count ()`, the average number of elements per bucket.

Example

```

// std_unordered_set_unordered_set_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_set::local_iterator

The type of a bucket iterator.

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

```

// std__unordered_set__unordered_set_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "]" ";

    return (0);
}

```

```

[c] [b] [a]
[a]

```

unordered_set::max_bucket_count

Gets the maximum number of buckets.

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

```
// std__unordered_set__unordered_set_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_set::max_load_factor

Gets or sets the maximum elements per bucket.

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example


```

// std__unordered_set__unordered_set_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_set::max_size

Gets the maximum size of the controlled sequence.

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

```

// std__unordered_set__unordered_set_max_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}

```

```
max_size() == 4294967295
```

unordered_set::operator=

Copies a hash table.

```

unordered_set& operator=(const unordered_set& right);

unordered_set& operator=(unordered_set&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>unordered_set</code> being copied into the <code>unordered_set</code> .

Remarks

After erasing any existing elements in an `unordered_set`, `operator=` either copies or moves the contents of *right* into the `unordered_set`.

Example

```
// unordered_set_operator_as.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

int main( )
{
    using namespace std;
    unordered_set<int> v1, v2, v3;
    unordered_set<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

unordered_set::pointer

The type of a pointer to an element.

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

```

// std__unordered_set__unordered_set_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::pointer p = &key;
        std::cout << "[" << *p << "]" ";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c] [b] [a]
```

unordered_set::reference

The type of a reference to an element.

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

```

// std__unordered_set__unordered_set_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::reference ref = key;
        std::cout << "[" << ref << "]" ";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c] [b] [a]
```

unordered_set::rehash

Rebuilds the hash table.

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

```

// std__unordered_set__unordered_set_rehash.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1

```

unordered_set::size

Counts the number of elements.

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

```
// std__unordered_set__unordered_set_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

unordered_set::size_type

The type of an unsigned distance between two elements.

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

```
// std__unordered_set__unordered_set_size_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;
    Myset::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

```
size == 0
```

unordered_set::swap

Swaps the contents of two containers.

```
void swap(unordered_set& right);
```

Parameters

right

The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `unordered_set::get_allocator()` `() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example


```

// std__unordered_set__unordered_set_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents "[f] [e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
[f] [e] [d]
[c] [b] [a]

```

unordered_set::unordered_set

Constructs a container object.

```

unordered_set(const unordered_set& Right);

explicit unordered_set(
    size_type bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

unordered_set(unordered_set&& Right);

unordered_set(initializer_list<Type> IList);

unordered_set(initializer_list<Type> IList, size_type bucket_count);

unordered_set(
    initializer_list<Type> IList,
    size_type bucket_count,
    const Hash& Hash);

unordered_set(
    initializer_list<Type> IList,
    size_type bucket_count,
    const Hash& Hash,
    const Comp& Comp);

unordered_set(
    initializer_list<Type> IList,
    size_type bucket_count,
    const Hash& Hash,
    const Comp& Comp,
    const Allocator& Al);

template <class InputIterator>
unordered_set(
    InputIterator first,
    InputIterator last,
    size_type bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

```

Parameters

PARAMETER	DESCRIPTION
<i>InputIterator</i>	The iterator type.
<i>Al</i>	The allocator object to store.
<i>Comp</i>	The comparison function object to store.
<i>Hash</i>	The hash function object to store.
<i>bucket_count</i>	The minimum number of buckets.
<i>Right</i>	The container to copy.
<i>IList</i>	The initializer_list containing the elements to copy.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor

specifies an empty controlled sequence. The third constructor specifies a copy of the sequence by moving *Right*. The fourth through eighth constructors use an `initializer_list` to specify the elements to copy. The ninth constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `NO`.

The hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Comp()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

unordered_set::value_type

The type of an element.

```
typedef Key value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

```
// std__unordered_set__unordered_set_value_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
[d] [c] [b] [a]
```

See also

[`<unordered_set>`](#)

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

unordered_multiset Class

11/9/2018 • 35 minutes to read • [Edit Online](#)

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

```
template <class Key,  
          class Hash = std::hash<Key>,  
          class Pred = std::equal_to<Key>,  
          class Alloc = std::allocator<Key>>  
class unordered_multiset;
```

Parameters

PARAMETER	DESCRIPTION
<i>Key</i>	The key type.
<i>Hash</i>	The hash function object type.
<i>Pred</i>	The equality comparison function object type.
<i>Alloc</i>	The allocator class.

Members

TYPE DEFINITION	DESCRIPTION
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.

TYPE DEFINITION	DESCRIPTION
difference_type	The type of a signed distance between two elements.
hasher	The type of the hash function.
iterator	The type of an iterator for the controlled sequence.
key_equal	The type of the comparison function.
key_type	The type of an ordering key.
local_iterator	The type of a bucket iterator for the controlled sequence.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.
MEMBER FUNCTION	DESCRIPTION
begin	Designates the beginning of the controlled sequence.
bucket	Gets the bucket number for a key value.
bucket_count	Gets the number of buckets.
bucket_size	Gets the size of a bucket.
cbegin	Designates the beginning of the controlled sequence.
cend	Designates the end of the controlled sequence.
clear	Removes all elements.
count	Finds the number of elements matching a specified key.
emplace	Adds an element constructed in place.
emplace_hint	Adds an element constructed in place, with hint.
empty	Tests whether no elements are present.
end	Designates the end of the controlled sequence.
equal_range	Finds range that matches a specified key.

MEMBER FUNCTION	DESCRIPTION
<code>erase</code>	Removes elements at specified positions.
<code>find</code>	Finds an element that matches a specified key.
<code>get_allocator</code>	Gets the stored allocator object.
<code>hash_function</code>	Gets the stored hash function object.
<code>insert</code>	Adds elements.
<code>key_eq</code>	Gets the stored comparison function object.
<code>load_factor</code>	Counts the average elements per bucket.
<code>max_bucket_count</code>	Gets the maximum number of buckets.
<code>max_load_factor</code>	Gets or sets the maximum elements per bucket.
<code>max_size</code>	Gets the maximum size of the controlled sequence.
<code>rehash</code>	Rebuilds the hash table.
<code>size</code>	Counts the number of elements.
<code>swap</code>	Swaps the contents of two containers.
<code>unordered_multiset</code>	Constructs a container object.

OPERATOR	DESCRIPTION
<code>unordered_multiset::operator=</code>	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `unordered_multiset::key_equal` and a hash function object of type `unordered_multiset::hasher`. You access the first stored object by calling the member function `unordered_multiset::key_eq()`; and you access the second stored object by calling the member function `unordered_multiset::hash_function()`. Specifically, for all values `x` and `y` of type `key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_set` Class, an object of template class `unordered_multiset` does not ensure that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys need not be unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `unordered_multiset::load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type `unordered_multiset::allocator_type`. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: `<unordered_set>`

Namespace: `std`

`unordered_multiset::allocator_type`

The type of an allocator for managing storage.

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

```
// std__unordered_set__unordered_multiset_allocator_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

`unordered_multiset::begin`

Designates the beginning of the controlled sequence or a bucket.


```

iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;

```

Parameters

PARAMETER	DESCRIPTION
<i>nbucket</i>	The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

```

// std__unordered_set__unordered_multiset_begin.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect first two items "[c] [b]"
    Myset::iterator it2 = c1.begin();
    std::cout << "[" << *it2 << "]" ";
    ++it2;
    std::cout << "[" << *it2 << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "]" ";

    return (0);
}

```

```

[c] [b] [a]
[c] [b]
[a]

```

unordered_multiset::bucket

Gets the bucket number for a key value.

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value `keyval`.

Example

```
// std__unordered_set__unordered_multiset_bucket.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multiset::bucket_count

Gets the number of buckets.

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

```
// std_unordered_set_unordered_multiset_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;

    return (0);
}
```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_multiset::bucket_size

Gets the size of a bucket

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

```

// std__unordered_set__unordered_multiset_bucket_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}

```

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multiset::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin` , the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator` . Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()` .

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator

auto i2 = Container.cbegin();
// i2 is Container<T>::const_iterator
```

unordered_multiset::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator` . Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()` .

```
auto i1 = Container.end();
// i1 is Container<T>::iterator

auto i2 = Container.cend();
// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

`unordered_multiset::clear`

Removes all elements.

```
void clear();
```

Remarks

The member function calls `unordered_multiset::erase` (`unordered_multiset::begin` (), `unordered_multiset::end` ()).

Example

```
// std__unordered_set__unordered_multiset_clear.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

unordered_multiset::const_iterator

The type of a constant iterator for the controlled sequence.

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

```
// std__unordered_set__unordered_multiset_const_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << " ] ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_multiset::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

```
// std__unordered_set__unordered_multiset_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "]" ";

    return (0);
}
```

```
[c] [b] [a]
[a]
```

unordered_multiset::const_pointer

The type of a constant pointer to an element.

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example


```

// std__unordered_set__unordered_multiset_const_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Myset::const_pointer p = &*it;
        std::cout << "[" << *p << "]" ";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c] [b] [a]
```

unordered_multiset::const_reference

The type of a constant reference to an element.

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

```
// std__unordered_set__unordered_multiset_const_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Myset::const_reference ref = *it;
        std::cout << "[" << ref << "]" ";
    }
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_multiset::count

Finds the number of elements matching a specified key.

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by [unordered_multiset::equal_range](#) (`keyval`) .

Example

```

// std__unordered_set__unordered_multiset_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}

```

```

[c] [b] [a]
count('A') == 0
count('b') == 1
count('C') == 0

```

unordered_multiset::difference_type

The type of a signed distance between two elements.

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

```

// std__unordered_set__unordered_multiset_difference_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // compute positive difference
    Myset::difference_type diff = 0;
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Myset::const_iterator it = c1.end();
         it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}

```

```

[c] [b] [a]
end()-begin() == 3
begin()-end() == -3

```

unordered_multiset::emplace

Inserts an element constructed in place (no copy or move operations are performed).

```

template <class... Args>
iterator emplace(Args&&... args);

```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered_multiset.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [multiset::emplace](#).

unordered_multiset::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

PARAMETER	DESCRIPTION
<i>args</i>	The arguments forwarded to construct an element to be inserted into the unordered_multiset.
<i>where</i>	A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see [set::emplace_hint](#).

unordered_multiset::empty

Tests whether no elements are present.

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

```

// std__unordered_set__unordered_multiset_empty.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}

```

```

[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false

```

unordered_multiset::end

Designates the end of the controlled sequence.

```

iterator end();
const_iterator end() const;
local_iterator end(size_type nbucket);
const_local_iterator end(size_type nbucket) const;

```

Parameters

nbucket

The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

```
// std__unordered_set__unordered_multiset_end.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect last two items "[a] [b]"
    Myset::iterator it2 = c1.end();
    --it2;
    std::cout << "[" << *it2 << "]" ";
    --it2;
    std::cout << "[" << *it2 << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << "[" << *lit << "]" ";

    return (0);
}
```

```
[c] [b] [a]
[a] [b]
[a]
```

unordered_multiset::equal_range

Finds range that matches a specified key.

```
std::pair<iterator, iterator>
    equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
    equal_range(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `x` such that `[x.first, x.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

```
// std__unordered_set__unordered_multiset_equal_range.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // display results of failed search
    std::pair<Myset::iterator, Myset::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "]" ";
    std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "]" ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
equal_range('x'):
equal_range('b'): [b]
```

unordered_multiset::erase

Removes an element or a range of elements in a `unordered_multiset` from specified positions or removes elements that match a specified key.


```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the `unordered_multiset` if no such element exists.

For the third member function, returns the number of elements that have been removed from the `unordered_multiset`.

Remarks

For a code example, see [set::erase](#).

unordered_multiset::find

Finds an element that matches a specified key.

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns [unordered_multiset::equal_range](#) `(keyval).first`.

Example

```

// std__unordered_set__unordered_multiset_find.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Myset::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << *it << "]" " << std::endl;

    return (0);
}

```

```

[c] [b] [a]
find('A') == false
find('b') == true: [b]

```

unordered_multiset::get_allocator

Gets the stored allocator object.

```

Alloc get_allocator() const;

```

Remarks

The member function returns the stored allocator object.

Example

```
// std__unordered_set__unordered_multiset_get_allocator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

```
al == std::allocator() is true
```

unordered_multiset::hash_function

Gets the stored hash function object.

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

```
// std__unordered_set__unordered_multiset_hash_function.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multiset::hasher

The type of the hash function.

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

```
// std__unordered_set__unordered_multiset_hasher.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multiset::insert

Inserts an element or a range of elements into an `unordered_multiset`.

```

// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
    IList);

```

Parameters

PARAMETER	DESCRIPTION
<i>Val</i>	The value of an element to be inserted into the <code>unordered_multiset</code> .
<i>Where</i>	The place to start searching for the correct point of insertion.
<i>ValTy</i>	Template parameter that specifies the argument type that the <code>unordered_multiset</code> can use to construct an element of value_type , and perfect-forwards <i>Val</i> as an argument.
<i>First</i>	The position of the first element to be copied.
<i>Last</i>	The position just beyond the last element to be copied.
<i>InputIterator</i>	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
<i>IList</i>	The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the

new element was inserted into the `unordered_multiset`.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `unordered_multiset`.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The `value_type` of a container is a typedef that belongs to the container, and, for set,

```
unordered_multiset<V>::value_type is type const V .
```

The range member function (5) inserts the sequence of element values into an `unordered_multiset` that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an `initializer_list` to copy elements into the `unordered_multiset`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see `unordered_multiset::emplace` and `unordered_multiset::emplace_hint`.

For a code example, see `multiset::insert`.

unordered_multiset::iterator

A type that provides a constant `forward iterator` that can read elements in an `unordered_multiset`.

```
typedef implementation-defined iterator;
```

Example

See the example for `begin` for an example of how to declare and use an **iterator**.

unordered_multiset::key_eq

Gets the stored comparison function object.

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```
// std__unordered_set__unordered_multiset_key_eq.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multiset::key_equal

The type of the comparison function.

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

```
// std__unordered_set__unordered_multiset_key_equal.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multiset::key_type

The type of an ordering key.

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

```
// std_unordered_set_unordered_multiset_key_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
[d] [c] [b] [a]
```

unordered_multiset::load_factor

Counts the average elements per bucket.

```
float load_factor() const;
```

Remarks

The member function returns `(float) unordered_multiset::size() / (float) unordered_multiset::bucket_count()`, the average number of elements per bucket.

Example


```

// std__unordered_set__unordered_multiset_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

unordered_multiset::local_iterator

The type of a bucket iterator.

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

```
// std__unordered_set__unordered_multiset_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "]" ";

    return (0);
}
```

```
[c] [b] [a]
[a]
```

unordered_multiset::max_bucket_count

Gets the maximum number of buckets.

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

```

// std__unordered_set__unordered_multiset_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multiset::max_load_factor

Gets or sets the maximum elements per bucket.

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example

```

// std__unordered_set__unordered_multiset_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1

```

unordered_multiset::max_size

Gets the maximum size of the controlled sequence.

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

```

// std__unordered_set__unordered_multiset_max_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}

```

```
max_size() == 4294967295
```

unordered_multiset::operator=

Copies a hash table.

```

unordered_multiset& operator=(const unordered_multiset& right);

unordered_multiset& operator=(unordered_multiset&& right);

```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The <code>unordered_multiset</code> being copied into the <code>unordered_multiset</code> .

Remarks

After erasing any existing elements in an `unordered_multiset`, `operator=` either copies or moves the contents of *right* into the `unordered_multiset`.

Example

```
// unordered_multiset_operator_as.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

int main( )
{
    using namespace std;
    unordered_multiset<int> v1, v2, v3;
    unordered_multiset<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

unordered_multiset::pointer

The type of a pointer to an element.

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

```

// std__unordered_set__unordered_multiset_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::pointer p = &key;
        std::cout << "[" << *p << "]" ";
    }
    std::cout << std::endl;

    return (0);
}

```

```
[c] [b] [a]
```

unordered_multiset::reference

The type of a reference to an element.

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example


```
// std__unordered_set__unordered_multiset_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
         it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::reference ref = key;
        std::cout << "[" << ref << "]" ";
    }
    std::cout << std::endl;

    return (0);
}
```

```
[c] [b] [a]
```

unordered_multiset::rehash

Rebuilds the hash table.

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

```

// std__unordered_set__unordered_multiset_rehash.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() << std::endl;

    return (0);
}

```

```

[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1

```

unordered_multiset::size

Counts the number of elements.

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

```
// std__unordered_set__unordered_multiset_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

unordered_multiset::size_type

The type of an unsigned distance between two elements.

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled

sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

```
// std__unordered_set__unordered_multiset_size_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;
    Myset::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

```
size == 0
```

unordered_multiset::swap

Swaps the contents of two containers.

```
void swap(unordered_multiset& right);
```

Parameters

right

The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `unordered_multiset::get_allocator() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

```

// std__unordered_set__unordered_multiset_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents "[f] [e] [d]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
[f] [e] [d]
[c] [b] [a]

```

unordered_multiset::unordered_multiset

Constructs a container object.

```

unordered_multiset(
    const unordered_multiset& Right);

explicit unordered_multiset(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

unordered_multiset(
    unordered_multiset&& Right);

unordered_set(
    initializer_list<Type> IList);

unordered_set(
    initializer_list<Typ> IList,
    size_type Bucket_count);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key,
    const Allocator& Al);

template <class InputIterator>
unordered_multiset(
    InputIterator First,
    InputIterator Last,
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

```

Parameters

PARAMETER	DESCRIPTION
<i>InputIterator</i>	The iterator type.
<i>Al</i>	The allocator object to store.
<i>Comp</i>	The comparison function object to store.
<i>Hash</i>	The hash function object to store.
<i>Bucket_count</i>	The minimum number of buckets.
<i>Right</i>	The container to copy.

PARAMETER	DESCRIPTION
<i>lList</i>	The initializer_list from which to copy.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[First, Last)`. The fourth constructor specifies a copy of the sequence by moving *Right*.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *Bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `N0`.

The hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Comp()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

unordered_multiset::value_type

The type of an element.

```
typedef Key value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

```

// std__unordered_set__unordered_multiset_value_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin();
         it != c1.end(); ++it)
        std::cout << "[" << *it << "]" ";
    std::cout << std::endl;

    return (0);
}

```

```

[c] [b] [a]
[d] [c] [b] [a]

```

See also

[<unordered_set>](#)

[Containers](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

<utility>

11/9/2018 • 2 minutes to read • [Edit Online](#)

Defines C++ Standard Library types, functions, and operators that help to construct and manage pairs of objects, which are useful whenever two objects need to be treated as if they were one.

Syntax

```
#include <utility>
```

Remarks

Pairs are widely used in the C++ Standard Library. They are required both as the arguments and return values for various functions and as element types for containers such as [map class](#) and [multimap class](#). The <utility> header is automatically included by <map> to assist in managing their key/value pair type elements.

Classes

CLASS	DESCRIPTION
tuple_element	A class that wraps the type of a <code>pair</code> element.
tuple_size	A class that wraps <code>pair</code> element count.

Functions

FUNCTION	DESCRIPTION
forward	Preserves the reference type (either <code>lvalue</code> or <code>rvalue</code>) of the argument from being obscured by perfect forwarding.
get	A function that gets an element from a <code>pair</code> object.
make_pair	A template helper function used to construct objects of type <code>pair</code> , where the component types are based on the data types passed as parameters.
move	Returns the passed in argument as an <code>rvalue</code> reference.
swap	Exchanges the elements of two <code>pair</code> objects.

Operators

OPERATOR	DESCRIPTION
operator!=	Tests if the pair object on the left side of the operator is not equal to the pair object on the right side.

OPERATOR	DESCRIPTION
<code>operator==</code>	Tests if the pair object on the left side of the operator is equal to the pair object on the right side.
<code>operator<</code>	Tests if the pair object on the left side of the operator is less than the pair object on the right side.
<code>operator<=</code>	Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.
<code>operator></code>	Tests if the pair object on the left side of the operator is greater than the pair object on the right side.
<code>operator>=</code>	Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

Structs

<code>identity</code>	
<code>pair</code>	A type that provides for the ability to treat two objects as a single object.

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<utility> functions

10/31/2018 • 7 minutes to read • [Edit Online](#)

exchange	forward	get Function <utility>
make_pair	move	swap

exchange

(C++14) Assigns a new value to an object and returns its old value.

```
template <class T, class Other = T>  
T exchange(T& val, Other&& new_val)
```

Parameters

val

The object that will receive the value of new_val.

new_val

The object whose value is copied or moved into val.

Remarks

For complex types, `exchange` avoids copying the old value when a move constructor is available, avoids copying the new value if it's a temporary object or is moved, and accepts any type as the new value, using any available converting assignment operator. The exchange function is different from `std::swap` in that the left argument is not moved or copied to the right argument.

Example

The following example shows how to use `exchange`. In the real world, `exchange` is most useful with large objects that are expensive to copy:

```

#include <utility>
#include <iostream>

using namespace std;

struct C
{
    int i;
    //...
};

int main()
{
    // Use brace initialization
    C c1{ 1 };
    C c2{ 2 };
    C result = exchange(c1, c2);
    cout << "The old value of c1 is: " << result.i << endl;
    cout << "The new value of c1 after exchange is: " << c1.i << endl;

    return 0;
}

```

```

The old value of c1 is: 1
The new value of c1 after exchange is: 2

```

forward

Conditionally casts its argument to an rvalue reference if the argument is an rvalue or rvalue reference. This restores the rvalue-ness of an argument to the forwarding function in support of perfect forwarding.

```

template <class Type>    // accepts lvalues
constexpr Type&& forward(typename remove_reference<Type>::type& Arg) noexcept

template <class Type>    // accepts everything else
constexpr Type&& forward(typename remove_reference<Type>::type&& Arg) noexcept

```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	The type of the value passed in <i>Arg</i> , which might be different than the type of <i>Arg</i> . Typically determined by a template argument of the forwarding function.
<i>Arg</i>	The argument to cast.

Return Value

Returns an rvalue reference to *Arg* if the value passed in *Arg* was originally an rvalue or a reference to an rvalue; otherwise, returns *Arg* without modifying its type.

Remarks

You must specify an explicit template argument to call `forward`.

`forward` does not forward its argument. Instead, by conditionally casting its argument to an rvalue reference if it was originally an rvalue or rvalue reference, `forward` enables the compiler to perform overload resolution with knowledge of the forwarded argument's original type. The apparent type of an argument to a forwarding function

might be different than its original type—for example, when an rvalue is used as an argument to a function and is bound to a parameter name; having a name makes it an lvalue, regardless of whether the value actually exists as an rvalue— `forward` restores the rvalue-ness of the argument.

Restoring the rvalue-ness of an argument's original value in order to perform overload resolution is known as *perfect forwarding*. Perfect forwarding enables a template function to accept an argument of either reference type and to restore its rvalue-ness when it's necessary for correct overload resolution. By using perfect forwarding, you can preserve move semantics for rvalues and avoid having to provide overloads for functions that vary only by the reference type of their arguments.

get

Gets an element from a `pair` object by index position, or by type.

```
// get reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
constexpr tuple_element_t<Index, pair<T1, T2>>&
get(pair<T1, T2>& Pr) noexcept;

// get reference to element T1 in pair Pr
template <class T1, class T2>
constexpr T1& get(pair<T1, T2>& Pr) noexcept;

// get reference to element T2 in pair Pr
template <class T2, class T1>
constexpr T2& get(pair<T1, T2>& Pr) noexcept;

// get const reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
constexpr const tuple_element_t<Index, pair<T1, T2>>&
get(const pair<T1, T2>& Pr) noexcept;

// get const reference to element T1 in pair Pr
template <class T1, class T2>
constexpr const T1& get(const pair<T1, T2>& Pr) noexcept;

// get const reference to element T2 in pair Pr
template <class T2, class T1>
constexpr const T2& get(const pair<T1, T2>& Pr) noexcept;

// get rvalue reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
constexpr tuple_element_t<Index, pair<T1, T2>>&&
get(pair<T1, T2>&& Pr) noexcept;

// get rvalue reference to element T1 in pair Pr
template <class T1, class T2>
constexpr T1&& get(pair<T1, T2>&& Pr) noexcept;

// get rvalue reference to element T2 in pair Pr
template <class T2, class T1>
constexpr T2&& get(pair<T1, T2>&& Pr) noexcept;
```

Parameters

Index

The 0-based index of the designated element.

T1

The type of the first pair element.

T2

The type of the second pair element.

pr

The pair to select from.

Remarks

The template functions each return a reference to an element of its `pair` argument.

For the indexed overloads, if the value of *Index* is 0 the functions return `pr.first` and if the value of *Index* is 1 the functions return `pr.second`. The type `RI` is the type of the returned element.

For the overloads that do not have an *Index* parameter, the element to return is deduced by the type argument. Calling `get<T>(Tuple)` will produce a compiler error if *pr* contains more or less than one element of type *T*.

Example

```
#include <utility>
#include <iostream>
using namespace std;
int main()
{

    typedef pair<int, double> MyPair;

    MyPair c0(9, 3.14);

    // get elements by index
    cout << " " << get<0>(c0);
    cout << " " << get<1>(c0) << endl;

    // get elements by type (C++14)
    MyPair c1(1, 0.27);
    cout << " " << get<int>(c1);
    cout << " " << get<double>(c1) << endl;

    /*
    Output:
    9 3.14
    1 0.27
    */

}
```

make_pair

A template function that you can use to construct objects of type `pair`, where the component types are automatically chosen based on the data types that are passed as parameters.

```
template <class T, class U>
pair<T, U> make_pair(T& Val1, U& Val2);

template <class T, class U>
pair<T, U> make_pair(T& Val1, U&& Val2);

template <class T, class U>
pair<T, U> make_pair(T&& Val1, U& Val2);

template <class T, class U>
pair<T, U> make_pair(T&& Val1, U&& Val2);
```

Parameters

Val1

Value that initializes the first element of `pair`.

`Val2`

Value that initializes the second element of `pair`.

Return Value

The pair object that's constructed: `pair < T, U >(Val1, Val2)`.

Remarks

`make_pair` converts object of type [reference_wrapper Class](#) to reference types and converts decaying arrays and functions to pointers.

In the returned `pair` object, `T` is determined as follows:

- If the input type `T` is `reference_wrapper<X>`, the returned type `T` is `X&`.
- Otherwise, the returned type `T` is `decay<T>::type`. If [decay Class](#) is not supported, the returned type `T` is the same as the input type `T`.

The returned type `U` is similarly determined from the input type `U`.

One advantage of `make_pair` is that the types of objects that are being stored are determined automatically by the compiler and do not have to be explicitly specified. Don't use explicit template arguments such as `make_pair<int, int>(1, 2)` when you use `make_pair` because it is unnecessarily verbose and adds complex rvalue reference problems that might cause compilation failure. For this example, the correct syntax would be

`make_pair(1, 2)`

The `make_pair` helper function also makes it possible to pass two values to a function that requires a pair as an input parameter.

Example

For an example about how to use the helper function `make_pair` to declare and initialize a pair, see [pair Structure](#).

move

Unconditionally casts its argument to an rvalue reference, and thereby signals that it can be moved if its type is move-enabled.

```
template <class Type>
constexpr typename remove_reference<Type>::type&& move(Type&& Arg) noexcept;
```

Parameters

PARAMETER	DESCRIPTION
<i>Type</i>	A type deduced from the type of the argument passed in <i>Arg</i> , together with the reference collapsing rules.
<i>Arg</i>	The argument to cast. Although the type of <i>Arg</i> appears to be specified as an rvalue reference, <code>move</code> also accepts lvalue arguments because lvalue references can bind to rvalue references.

Return Value

`Arg` as an rvalue reference, whether or not its type is a reference type.

Remarks

The template argument *Type* is not intended to be specified explicitly, but to be deduced from the type of the value passed in *Arg*. The type of *Type* is further adjusted according to the reference collapsing rules.

`move` does not move its argument. Instead, by unconditionally casting its argument—which might be an lvalue—to an rvalue reference, it enables the compiler to subsequently move, rather than copy, the value passed in *Arg* if its type is move-enabled. If its type is not move-enabled, it is copied instead.

If the value passed in *Arg* is an lvalue—that is, it has a name or its address can be taken—it's invalidated when the move occurs. Do not refer to the value passed in *Arg* by its name or address after it's been moved.

swap

Exchanges the elements of two [pair Structure](#) objects.

```
template <class T, class U>
void swap(pair<T, U>& left, pair<T, U>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>pair</code> .
<i>right</i>	An object of type <code>pair</code> .

Remarks

One advantage of `swap` is that the types of objects that are being stored are determined automatically by the compiler and do not have to be explicitly specified. Don't use explicit template arguments such as

`swap<int, int>(1, 2)` when you use `swap` because it is unnecessarily verbose and adds complex rvalue reference problems that might cause compilation failure.

See also

[<utility>](#)

<utility> operators

10/31/2018 • 12 minutes to read • [Edit Online](#)

operator!=	operator>	operator>=
operator<	operator<=	operator==

operator!=

Tests if the pair object on the left side of the operator is not equal to the pair object on the right side.

```
template <class Type>
constexpr bool operator!=(const Type& left, const Type& right);

template <class T, class U>
constexpr bool operator!=(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left

An object of type `pair`.

right

An object of type `pair`.

Return Value

true if the pairs are not equal; **false** if the pairs are equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

```

// utility_op_ne.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3;

    p1 = make_pair ( 10, 1.11e-1 );
    p2 = make_pair ( 1000, 1.11e-3 );
    p3 = make_pair ( 10, 1.11e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

    if ( p1 != p2 )
        cout << "The pairs p1 and p2 are not equal." << endl;
    else
        cout << "The pairs p1 and p2 are equal." << endl;

    if ( p1 != p3 )
        cout << "The pairs p1 and p3 are not equal." << endl;
    else
        cout << "The pairs p1 and p3 are equal." << endl;
}

```

```

The pair p1 is: ( 10, 0.111 ).
The pair p2 is: ( 1000, 0.00111 ).
The pair p3 is: ( 10, 0.111 ).

The pairs p1 and p2 are not equal.
The pairs p1 and p3 are equal.

```

operator==

Tests if the pair object on the left side of the operator is equal to the pair object on the right side.

```

template <class T, class U>
constexpr bool operator==(const pair<T, U>& left, const pair<T, U>& right);

```

Parameters

left

An object of type `pair`.

right

An object of type `pair`.

Return Value

true if the pairs are equal; **false** if the `pair`s are not equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. The function returns `left.first == right.first && left.second == right.second`. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

```
// utility_op_eq.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3;

    p1 = make_pair ( 10, 1.11e-1 );
    p2 = make_pair ( 1000, 1.11e-3 );
    p3 = make_pair ( 10, 1.11e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

    if ( p1 == p2 )
        cout << "The pairs p1 and p2 are equal." << endl;
    else
        cout << "The pairs p1 and p2 are not equal." << endl;

    if ( p1 == p3 )
        cout << "The pairs p1 and p3 are equal." << endl;
    else
        cout << "The pairs p1 and p3 are not equal." << endl;
}
```

operator<

Tests if the pair object on the left side of the operator is less than the pair object on the right side.

```
template <class T, class U>
constexpr bool operator<(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left

An object of type `pair` on the left side of the operator.

right

An object of type `pair` on the right side of the operator.

Return Value

true if the `pair` on the left side of the operator is strictly less than the `pair` on the right side of the operator; otherwise **false**.

Remarks

The `left` `pair` object is said to be strictly less than the `right` `pair` object if *left* is less than and not equal to

right.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_lt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3;

    p1 = make_pair ( 10, 2.22e-1 );
    p2 = make_pair ( 100, 1.11e-1 );
    p3 = make_pair ( 10, 1.11e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

    if ( p1 < p2 )
        cout << "The pair p1 is less than the pair p2." << endl;
    else
        cout << "The pair p1 is not less than the pair p2." << endl;

    if ( p1 < p3 )
        cout << "The pair p1 is less than the pair p3." << endl;
    else
        cout << "The pair p1 is not less than the pair p3." << endl;
}
```

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
```

```
The pair p1 is less than the pair p2.
The pair p1 is not less than the pair p3.
```

operator<=

Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.

```
template <class Type>
constexpr bool operator<=(const Type& left, const Type& right);

template <class T, class U>
constexpr bool operator<=(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left

An object of type `pair` on the left side of the operator.

right

An object of type `pair` on the right side of the operator.

Return Value

true if the `pair` on the left side of the operator is less than or equal to the `pair` on the right side of the operator; otherwise **false**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_le.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3, p4;

    p1 = make_pair ( 10, 2.22e-1 );
    p2 = make_pair ( 100, 1.11e-1 );
    p3 = make_pair ( 10, 1.11e-1 );
    p4 = make_pair ( 10, 2.22e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
    cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

    if ( p1 <= p2 )
        cout << "The pair p1 is less than or equal to the pair p2." << endl;
    else
        cout << "The pair p1 is greater than the pair p2." << endl;

    if ( p1 <= p3 )
        cout << "The pair p1 is less than or equal to the pair p3." << endl;
    else
        cout << "The pair p1 is greater than the pair p3." << endl;

    if ( p1 <= p4 )
        cout << "The pair p1 is less than or equal to the pair p4." << endl;
    else
        cout << "The pair p1 is greater than the pair p4." << endl;
}
```

```
The pair p1 is: ( 10, 0.222 ).  
The pair p2 is: ( 100, 0.111 ).  
The pair p3 is: ( 10, 0.111 ).  
The pair p4 is: ( 10, 0.222 ).
```

```
The pair p1 is less than or equal to the pair p2.  
The pair p1 is greater than the pair p3.  
The pair p1 is less than or equal to the pair p4.
```

operator>

Tests if the pair object on the left side of the operator is greater than the pair object on the right side.

```
template <class Type>  
constexpr bool operator>(const Type& left, const Type& right);  
  
template <class T, class U>  
constexpr bool operator>(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left

An object of type `pair` on the left side of the operator.

right

An object of type `pair` on the right side of the operator.

Return Value

true if the `pair` on the left side of the operator is strictly greater than the `pair` on the right side of the operator; otherwise **false**.

Remarks

The `left` `pair` object is said to be strictly greater than the `right` `pair` object if *left* is greater than and not equal to *right*.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```

// utility_op_gt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3, p4;

    p1 = make_pair ( 10, 2.22e-1 );
    p2 = make_pair ( 100, 1.11e-1 );
    p3 = make_pair ( 10, 1.11e-1 );
    p4 = make_pair ( 10, 2.22e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
    cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

    if ( p1 > p2 )
        cout << "The pair p1 is greater than the pair p2." << endl;
    else
        cout << "The pair p1 is not greater than the pair p2." << endl;

    if ( p1 > p3 )
        cout << "The pair p1 is greater than the pair p3." << endl;
    else
        cout << "The pair p1 is not greater than the pair p3." << endl;

    if ( p1 > p4 )
        cout << "The pair p1 is greater than the pair p4." << endl;
    else
        cout << "The pair p1 is not greater than the pair p4." << endl;
}

```

```

The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).

```

```

The pair p1 is not greater than the pair p2.
The pair p1 is greater than the pair p3.
The pair p1 is not greater than the pair p4.

```

operator>=

Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

```

template <class Type>
constexpr bool operator>=(const Type& left, const Type& right);

template <class T, class U>
constexpr bool operator>=(const pair<T, U>& left, const pair<T, U>& right);

```

Parameters

left

An object of type `pair` on the left side of the operator.

right

An object of type `pair` on the right side of the operator.

Return Value

true if the `pair` on the left side of the operator is greater than or equal to the `pair` on the right side of the operator; otherwise **false**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

```
// utility_op_ge.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;
    pair <int, double> p1, p2, p3, p4;

    p1 = make_pair ( 10, 2.22e-1 );
    p2 = make_pair ( 100, 1.11e-1 );
    p3 = make_pair ( 10, 1.11e-1 );
    p4 = make_pair ( 10, 2.22e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
    cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

    if ( p1 >= p2 )
        cout << "Pair p1 is greater than or equal to pair p2." << endl;
    else
        cout << "The pair p1 is less than the pair p2." << endl;

    if ( p1 >= p3 )
        cout << "Pair p1 is greater than or equal to pair p3." << endl;
    else
        cout << "The pair p1 is less than the pair p3." << endl;

    if ( p1 >= p4 )
        cout << "Pair p1 is greater than or equal to pair p4." << endl;
    else
        cout << "The pair p1 is less than the pair p4." << endl;
}
```



```
The pair p1 is: ( 10, 0.222 ).  
The pair p2 is: ( 100, 0.111 ).  
The pair p3 is: ( 10, 0.111 ).  
The pair p4 is: ( 10, 0.222 ).
```

```
The pair p1 is less than the pair p2.  
Pair p1 is greater than or equal to pair p3.  
Pair p1 is greater than or equal to pair p4.
```

See also

[<utility>](#)

identity Structure

10/31/2018 • 2 minutes to read • [Edit Online](#)

A struct that provides a type definition as the template parameter.

Syntax

```
struct identity {  
    typedef Type type;  
    Type operator()(const Type& left) const;  
};
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	The value to identify.

Remarks

The class contains the public type definition `type`, which is the same as the template parameter `Type`. It is used in conjunction with template function [forward](#) to ensure that a function parameter has the desired type.

For compatibility with older code, the class also defines the identity function `operator()` which returns its argument *left*.

Requirements

Header: <utility>

Namespace: std

See also

[<utility>](#)

pair Structure

10/31/2018 • 3 minutes to read • [Edit Online](#)

A struct that provides for the ability to treat two objects as a single object.

Syntax

```
struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    constexpr pair();
    constexpr pair(
        const T1& Val1,
        const T2& Val2);

    template <class Other1, class Other2>
    constexpr pair(const pair<Other1, Other2>& Right);

    template <class Other1, class Other2>
    constexpr pair(const pair <Other1 Val1, Other2 Val2>&& Right);

    template <class Other1, class Other2>
    constexpr pair(Other1&& Val1, Other2&& Val2);
};
```

Parameters

Val1

Value initializing the first element of `pair`.

Val2

Value initializing the second element of `pair`.

Right

A pair whose values are to be used to initialize the elements of another pair.

Return Value

The first (default) constructor initializes first element of the pair to the default of type `T1` and second element to default of type `T2`.

The second constructor initializes first element of the pair to *Val1* and second to *Val2*.

The third (template) constructor initializes first element of the pair to `Right`, **first** and second to `Right`, **second**.

The fourth constructor initializes first element of the pair to *Val1* and second to *Val2* using [Rvalue Reference Declarator: &&](#).

Remarks

The template struct stores a pair of objects of type `T1` and `T2`, respectively. The type `first_type` is the same as the template parameter `T1` and the type `second_type` is the same as the template parameter `T2`. `T1` and `T2`

each need supply only a default constructor, a single-argument constructor, and a destructor. All members of the type `pair` are public, because the type is declared as a `struct` rather than as a **class**. The two most common uses for a pair are as return types for functions that return two values and as elements for the associative container classes [map Class](#) and [multimap Class](#) that have both a key and a value type associated with each element. The latter satisfies the requirements for a pair associative container and has a value type of the form `pair < const key_type, mapped_type >`.

Example

```
// utility_pair.cpp
// compile with: /EHsc
#include <utility>
#include <map>
#include <iomanip>
#include <iostream>

int main( )
{
    using namespace std;

    // Using the constructor to declare and initialize a pair
    pair <int, double> p1 ( 10, 1.1e-2 );

    // Compare using the helper function to declare and initialize a pair
    pair <int, double> p2;
    p2 = make_pair ( 10, 2.22e-1 );

    // Making a copy of a pair
    pair <int, double> p3 ( p1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
          << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
          << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
          << p3.second << " )." << endl;

    // Using a pair for a map element
    map <int, int> m1;
    map <int, int>::iterator m1_Iter;

    typedef pair <int, int> Map_Int_Pair;

    m1.insert ( Map_Int_Pair ( 1, 10 ) );
    m1.insert ( Map_Int_Pair ( 2, 20 ) );
    m1.insert ( Map_Int_Pair ( 3, 30 ) );

    cout << "The element pairs of the map m1 are:";
    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
        cout << " ( " << m1_Iter->first << ", "
              << m1_Iter->second << " )";
    cout << "." << endl;

    // Using pair as a return type for a function
    pair< map<int,int>::iterator, bool > pr1, pr2;
    pr1 = m1.insert ( Map_Int_Pair ( 4, 40 ) );
    pr2 = m1.insert ( Map_Int_Pair ( 1, 10 ) );

    if( pr1.second == true )
    {
        cout << "The element (4,40) was inserted successfully in m1."
              << endl;
    }
    else
```

```

{
    cout << "The element with a key value of\n"
        << " ( (pr1.first) -> first ) = " << ( pr1.first ) -> first
        << " is already in m1,\n so the insertion failed." << endl;
}

if( pr2.second == true )
{
    cout << "The element (1,10) was inserted successfully in m1."
        << endl;
}
else
{
    cout << "The element with a key value of\n"
        << " ( (pr2.first) -> first ) = " << ( pr2.first ) -> first
        << " is already in m1,\n so the insertion failed." << endl;
}
}
/* Output:
The pair p1 is: ( 10, 0.011 ).
The pair p2 is: ( 10, 0.222 ).
The pair p3 is: ( 10, 0.011 ).
The element pairs of the map m1 are: ( 1, 10 ) ( 2, 20 ) ( 3, 30 ).
The element (4,40) was inserted successfully in m1.
The element with a key value of
( (pr2.first) -> first ) = 1 is already in m1,
so the insertion failed.
*/

```

Requirements

Header: <utility>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

<valarray>

11/8/2018 • 6 minutes to read • [Edit Online](#)

Defines the template class `valarray` and numerous supporting template classes and functions.

Syntax

```
#include <valarray>
```

Remarks

These template classes and functions are permitted unusual latitude in the interest of improved performance. Specifically, any function returning type `valarray<T1>` may return an object of some other type `T2`. In that case, any function that accepts one or more arguments of type `valarray<T2>` must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type `T2`.

Functions

FUNCTION	DESCRIPTION
abs	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the absolute value of the elements of the input <code>valarray</code> .
acos	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the arccosine of the elements of the input <code>valarray</code> .
asin	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the arcsine of the elements of the input <code>valarray</code> .
atan	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the principal value of the arctangent of the elements of the input <code>valarray</code> .
atan2	Returns a <code>valarray</code> whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of <code>valarrays</code> .
cos	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the cosine of the elements of the input <code>valarray</code> .
cosh	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the hyperbolic cosine of the elements of the input <code>valarray</code> .
exp	Operates on the elements of an input <code>valarray</code> , returning a <code>valarray</code> whose elements are equal to the natural exponential of the elements of the input <code>valarray</code> .

FUNCTION	DESCRIPTION
<code>log</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural logarithm of the elements of the input valarray.
<code>log10</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the base 10 or common logarithm of the elements of the input valarray.
<code>pow</code>	Operates on the elements of input valarrays and constants, returning a valarray whose elements are equal to a base specified either by the elements of an input valarray or a constant raised to an exponent specified either by the elements of an input valarray or a constant.
<code>sin</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the sine of the elements of the input valarray.
<code>sinh</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.
<code>sqrt</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the square root of the elements of the input valarray.
<code>swap</code>	
<code>tan</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the tangent of the elements of the input valarray.
<code>tanh</code>	Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic tangent of the elements of the input valarray.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests whether the corresponding elements of two equally sized valarrays are unequal or whether all the elements of a valarray are unequal a specified value of the valarray's element type.
<code>operator%</code>	Obtains the remainder of dividing the corresponding elements of two equally sized valarrays or of dividing a valarray by a specified value of the valarray's element type or of dividing a specified value by a valarray.
<code>operator&</code>	Obtains the bitwise <code>AND</code> between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

OPERATOR	DESCRIPTION
<code>operator&&</code>	Obtains the logical <code>AND</code> between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.
<code>operator></code>	Tests whether the elements of one valarray are greater than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value of the valarray's element type.
<code>operator>=</code>	Tests whether the elements of one valarray are greater than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.
<code>operator>></code>	Right-shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.
<code>operator<</code>	Tests whether the elements of one valarray are less than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.
<code>operator<=</code>	Tests whether the elements of one valarray are less than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.
<code>operator<<</code>	Left shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.
<code>operator*</code>	Obtains the element-wise product between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
<code>operator+</code>	Obtains the element-wise sum between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
<code>operator-</code>	Obtains the element-wise difference between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
<code>operator/</code>	Obtains the element-wise quotient between corresponding elements of two equally sized valarrays or of between a valarray a specified value of the valarray's element type.
<code>operator==</code>	Tests whether the corresponding elements of two equally sized valarrays are equal or whether all the elements of a valarray are equal a specified value of the valarray's element type.
<code>operator^</code>	Obtains the bitwise exclusive <code>OR</code> between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

OPERATOR	DESCRIPTION
operator 	Obtains the bitwise <code>OR</code> between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.
operator 	Obtains the logical <code>OR</code> between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.

Classes

CLASS	DESCRIPTION
gslice Class	A utility class to valarray that is used to define multi-dimensional slices of a valarray.
gslice_array Class	An internal, auxiliary template class that supports general slice objects by providing operations between subset arrays defined by the general slice of a valarray.
indirect_array Class	An internal, auxiliary template class that supports objects that are subsets of valarrays by providing operations between subset arrays defined by specifying a subset of indices of a parent valarray.
mask_array Class	An internal, auxiliary template class that supports objects that are subsets of parent valarrays, specified with a Boolean expression, by providing operations between the subset arrays.
slice Class	A utility class to valarray that is used to define one-dimensional, vector-like subsets of a valarray.
slice_array Class	An internal, auxiliary template class that supports slice objects by providing operations between subset arrays defined by the slice of a valarray.
valarray Class	The template class describes an object that controls a sequence of elements of type <code>Type</code> that are stored as an array and designed for performing high-speed mathematical operations, optimized for computational performance.

Specializations

valarray<bool> Class	A specialized version of the template class <code>valarray<Type></code> to elements of type bool .
--	---

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

<valarray> functions

10/31/2018 • 22 minutes to read • [Edit Online](#)

abs	acos	asin
atan	atan2	cos
cosh	exp	log
log10	pow	sin
sinh	sqrt	swap
tan	tanh	

abs

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the absolute value of the elements of the input valarray.

```
template <class Type>
valarray<Type> abs(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```

// valarray_abs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va1 ( 9 ), va2 ( 9 );
    for ( i = 0 ; i < 4 ; i++ )
        va1 [ i ] = -i;
    for ( i = 4 ; i < 9 ; i++ )
        va1 [ i ] = i;

    cout << "The initial valarray is: ";
    for (i = 0 ; i < 9 ; i++ )
        cout << va1 [ i ] << " ";
    cout << "." << endl;

    va2 = abs ( va1 );
    cout << "The absolute value of the initial valarray is: ";
    for (i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << " ";
    cout << "." << endl;
}

```

```

The initial valarray is: 0 -1 -2 -3 4 5 6 7 8 .
The absolute value of the initial valarray is: 0 1 2 3 4 5 6 7 8 .

```

acos

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arccosine of the elements of the input valarray.

```

template <class Type>
valarray<Type> acos(const valarray<Type>& left);

```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arccosine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between 0 and +pi that is consistent with the cosine value input.

Example

```

// valarray_acos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = 0.25 * i - 1;
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:";
    for (i = 0 ; i < 9 ; i++ )
        cout << " " << va1 [ i ];
    cout << endl;

    va2 = acos ( va1 );
    cout << "The arccosine of the initial valarray is:\n";
    for (i = 0 ; i < 9 ; i++ )
        cout << setw(10) << va2 [ i ]
            << " radians, which is "
            << setw(11) << (180/pi) * va2 [ i ]
            << " degrees" << endl;
}

```

```

The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arccosine of the initial valarray is:
 3.14159 radians, which is      180 degrees
 2.41886 radians, which is     138.59 degrees
 2.0944 radians, which is      120 degrees
 1.82348 radians, which is     104.478 degrees
 1.5708 radians, which is       90 degrees
 1.31812 radians, which is      75.5225 degrees
 1.0472 radians, which is       60 degrees
 0.722734 radians, which is     41.4096 degrees
 0 radians, which is           0 degrees

```

asin

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arcsine of the elements of the input valarray.

```

template <class Type>
valarray<Type> asin(const valarray<Type>& left);

```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arcsine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between $+\pi/2$ and $-\pi/2$ that is consistent with the sine value input.

Example

```
// valarray_asin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = 0.25 * i - 1;
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:";
    for ( i = 0 ; i < 9 ; i++ )
        cout << " " << va1 [ i ];
    cout << endl;

    va2 = asin ( va1 );
    cout << "The arcsine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw(10) << va2 [ i ]
            << " radians, which is "
            << setw(11) << (180/pi) * va2 [ i ]
            << " degrees" << endl;
}
```

```
The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arcsine of the initial valarray is:
-1.5708 radians, which is -90 degrees
-0.848062 radians, which is -48.5904 degrees
-0.523599 radians, which is -30 degrees
-0.25268 radians, which is -14.4775 degrees
0 radians, which is 0 degrees
0.25268 radians, which is 14.4775 degrees
0.523599 radians, which is 30 degrees
0.848062 radians, which is 48.5904 degrees
1.5708 radians, which is 90 degrees
```

atan

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the principal value of the arctangent of the elements of the input valarray.

```
template <class Type>
valarray<Type> atan(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arctangent of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between $+\pi/2$ and $-\pi/2$ that is consistent with the tangent value input.

Example

```
// valarray_atan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    va1 [ 0 ] = -100;
    for ( i = 1 ; i < 8 ; i++ )
        va1 [ i ] = 5 * ( 0.25 * i - 1 );
    va1 [ 8 ] = 100;
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is: ";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va1 [ i ] << " ";
    cout << "." << endl;

    va2 = atan ( va1 );
    cout << "The arcsine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw(10) << va2 [ i ]
            << " radians, which is "
            << setw(11) << (180/pi) * va2 [ i ]
            << " degrees" << endl;
    cout << endl;
}
```

```
The initial valarray is: -100 -3.75 -2.5 -1.25 0 1.25 2.5 3.75 100 .
The arcsine of the initial valarray is:
-1.5608 radians, which is -89.4271 degrees
-1.31019 radians, which is -75.0686 degrees
-1.19029 radians, which is -68.1986 degrees
-0.896055 radians, which is -51.3402 degrees
0 radians, which is 0 degrees
0.896055 radians, which is 51.3402 degrees
1.19029 radians, which is 68.1986 degrees
1.31019 radians, which is 75.0686 degrees
1.5608 radians, which is 89.4271 degrees
```

atan2

Returns a valarray whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of valarrays.

```
template <class Type>
valarray<Type> atan2(const valarray<Type>& left, const valarray<Type>& right);

template <class Type>
valarray<Type> atan2(const valarray<Type> left, const Type& right);

template <class Type>
valarray<Type> atan2(const Type& left, const valarray<Type>& right);
```

Parameters

left

The constant numerical data type or input valarray whose elements provide the values for the y-coordinate of the arctangent argument.

right

The constant numerical data type or input valarray whose elements provide the values for the x-coordinate of the arctangent argument.

Return Value

A valarray whose elements `I` are equal to the arctangent of:

- `left [I] / _Right [I]` for the first template function.
- `left [I] / right` for the second template function.
- `left / right [I]` for the third template function.

Remarks

The units of the returned elements are in radians.

This function preserves information about the signs of the components in the argument that is lost by the standard tangent function, and this knowledge of the quadrant enables the return value to be assigned a unique angle between +pi and -pi.

If *left* and *right* have a different number of elements, the result is undefined.

Example

```

// valarray_atan2.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1y ( 1 , 4 ), va1x ( 1 , 4 );
    va1x [ 1 ] = -1;
    va1x [ 2 ] = -1;
    va1y [ 2 ] = -1;
    va1y [ 3 ] = -1;
    valarray<double> va2 ( 4 );

    cout << "The initial valarray for the x coordinate is: ( ";
    for ( i = 0 ; i < 4 ; i++ )
        cout << va1x [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial valarray for the y coordinate is: ( ";
    for ( i = 0 ; i < 4 ; i++ )
        cout << va1y [ i ] << " ";
    cout << ")." << endl;

    va2 = atan2 ( va1y , va1x );
    cout << "The atan2 ( y / x ) of the initial valarrays is:\n";
    for ( i = 0 ; i < 4 ; i++ )
        cout << setw( 10 ) << va2 [ i ]
            << " radians, which is "
            << setw( 11 ) << ( 180/pi ) * va2 [ i ]
            << "degrees" << endl;
    cout << endl;
}

```

```

The initial valarray for the x coordinate is: ( 1 -1 -1 1 ).
The initial valarray for the y coordinate is: ( 1 1 -1 -1 ).
The atan2 ( y / x ) of the initial valarrays is:
0.785398 radians, which is      45degrees
2.35619 radians, which is     135degrees
-2.35619 radians, which is    -135degrees
-0.785398 radians, which is    -45degrees

```

COS

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the cosine of the elements of the input valarray.

```

template <class Type>
valarray<Type> cos(const valarray<Type>& left);

```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```
// valarray_cos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = ( pi ) * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw( 10 ) << va1 [ i ]
            << " radians, which is "
            << setw( 5 ) << ( 180/pi ) * va1 [ i ]
            << " degrees" << endl;
    cout << endl;

    va2 = cos ( va1 );
    cout << "The cosine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

```
The initial valarray is:
-3.14159 radians, which is -180 degrees
-2.35619 radians, which is -135 degrees
-1.5708 radians, which is -90 degrees
-0.785398 radians, which is -45 degrees
0 radians, which is 0 degrees
0.785398 radians, which is 45 degrees
1.5708 radians, which is 90 degrees
2.35619 radians, which is 135 degrees
3.14159 radians, which is 180 degrees
```

```
The cosine of the initial valarray is:
-1
-0.707107
-1.03412e-013
0.707107
1
0.707107
-1.03412e-013
-0.707107
-1
```

cosh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

```
template <class Type>
valarray<Type> cosh(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic cosine in terms of exponential function:

$$\cosh(z) = (\exp(z) + \exp(-z)) / 2$$

Example

```
// valarray_cosh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = pi * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw( 10 ) << va1 [ i ]
        << " radians, which is "
        << setw( 5 ) << ( 180/pi ) * va1 [ i ]
        << " degrees" << endl;
    cout << endl;

    va2 = cosh ( va1 );
    cout << "The hyperbolic cosine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

The initial valarray is:

-3.14159	radians, which is	-180	degrees
-2.35619	radians, which is	-135	degrees
-1.5708	radians, which is	-90	degrees
-0.785398	radians, which is	-45	degrees
0	radians, which is	0	degrees
0.785398	radians, which is	45	degrees
1.5708	radians, which is	90	degrees
2.35619	radians, which is	135	degrees
3.14159	radians, which is	180	degrees

The hyperbolic cosine of the initial valarray is:

11.592
5.32275
2.50918
1.32461
1
1.32461
2.50918
5.32275
11.592

exp

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural exponential of the elements of the input valarray.

```
template <class Type>
valarray<Type> exp(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the natural exponential of the elements of the input valarray.

Example

```

// valarray_exp.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = 10 * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "Initial valarray:";
    for ( i = 0 ; i < 9 ; i++ )
        cout << " " << va1 [ i ];
    cout << endl;

    va2 = exp ( va1 );
    cout << "The natural exponential of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}

```

```

Initial valarray: -10 -7.5 -5 -2.5 0 2.5 5 7.5 10
The natural exponential of the initial valarray is:
4.53999e-005
0.000553084
0.00673795
0.082085
1
12.1825
148.413
1808.04
22026.5

```

log

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural logarithm of the elements of the input valarray.

```

template <class Type>
valarray<Type> log(const valarray<Type>& left);

```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

```
// valarray_log.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = 10 * i;
    valarray<double> va2 ( 9 );

    cout << "Initial valarray:";
    for ( i = 0 ; i < 9 ; i++ )
        cout << " " << va1 [ i ];
    cout << endl;

    va2 = log ( va1 );
    cout << "The natural logarithm of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

```
Initial valarray: 0 10 20 30 40 50 60 70 80
The natural logarithm of the initial valarray is:
-inf
2.30259
2.99573
3.4012
3.68888
3.91202
4.09434
4.2485
4.38203
```

log10

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the base 10 or common logarithm of the elements of the input valarray.

```
template <class Type>
valarray<Type> log10(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the common logarithm of the elements of the input valarray.

Example

```

// valarray_log10.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    int i;

    valarray<double> va1 ( 11 );
    for ( i = 0 ; i < 11 ; i++ )
        va1 [ i ] = 10 * i;
    valarray<double> va2 ( 9 );

    cout << "Initial valarray:";
    for (i = 0 ; i < 11 ; i++ )
        cout << " " << va1 [ i ];
    cout << endl;

    va2 = log10 ( va1 );
    cout << "The common logarithm of the initial valarray is:\n";
    for (i = 0 ; i < 11 ; i++ )
        cout << va2 [ i ] << endl;
}

```

```

Initial valarray: 0 10 20 30 40 50 60 70 80 90 100
The common logarithm of the initial valarray is:
-inf
1
1.30103
1.47712
1.60206
1.69897
1.77815
1.8451
1.90309
1.95424
2

```

pow

Operates on the elements of input valarrays and constants, returning a valarray whose elements are equal to a base specified either by the elements of an input valarray or a constant raised to an exponent specified either by the elements of an input valarray or a constant.

```

template <class Type>
valarray<Type>
pow(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
pow(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
pow(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The input valarray whose elements supply the base for each element to be exponentiated.

right

The input valarray whose elements supply the power for each element to be exponentiated.

Return Value

A valarray whose elements `I` are equal to:

- `left` `[/]` raised to the power `right` `[/]` for the first template function.
- `left` `[/]` raised to the power `right` for the second template function.
- `left` raised to the power `right` `[/]` for the third template function.

Remarks

If *left* and *right* have a different number of elements, the result is undefined.

Example

```

#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> vabase ( 6 );
    for ( i = 0 ; i < 6 ; i++ )
        vabase [ i ] = i/2;
    valarray<double> vaexp ( 6 );
    for ( i = 0 ; i < 6 ; i++ )
        vaexp [ i ] = 2 * i;

    valarray<double> va2 ( 6 );

    cout << "The initial valarray for the base is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vabase [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial valarray for the exponent is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaexp[ i ] << " ";
    cout << ")." << endl;

    va2 = pow ( vabase , vaexp );
    cout << "The power of (n/2) * exp (2n) for n = 0 to n = 5 is: \n";
    for ( i = 0 ; i < 6 ; i++ )
        cout << "n = " << i << "\tgives " << va2 [ i ] << endl;
}

```

```

The initial valarray for the base is: ( 0 0 1 1 2 2 ).
The initial valarray for the exponent is: ( 0 2 4 6 8 10 ).
The power of (n/2) * exp (2n) for n = 0 to n = 5 is:
n = 0   gives 1
n = 1   gives 0
n = 2   gives 1
n = 3   gives 1
n = 4   gives 256
n = 5   gives 1024

```

sin

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the sine of the elements of the input valarray.

```

template <class Type>
valarray<Type> sin(const valarray<Type>& left);

```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the sine of the elements of the input valarray.

Example

```
// valarray_sin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = pi * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw(10) << va1 [ i ]
            << " radians, which is "
            << setw(5) << ( 180/pi ) * va1 [ i ]
            << " degrees" << endl;
    cout << endl;

    va2 = sin ( va1 );
    cout << "The sine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

```
The initial valarray is:
-3.14159 radians, which is -180 degrees
-2.35619 radians, which is -135 degrees
-1.5708 radians, which is -90 degrees
-0.785398 radians, which is -45 degrees
0 radians, which is 0 degrees
0.785398 radians, which is 45 degrees
1.5708 radians, which is 90 degrees
2.35619 radians, which is 135 degrees
3.14159 radians, which is 180 degrees
```

```
The sine of the initial valarray is:
2.06823e-013
-0.707107
-1
-0.707107
0
0.707107
1
0.707107
-2.06823e-013
```

sinh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

```
template <class Type>
valarray<Type> sinh(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic sine in terms of exponential function:

$$\sinh (z) = (\exp (z) - \exp (- z)) / 2$$

Example

```
// valarray_sinh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = pi * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw( 10 ) << va1 [ i ]
            << " radians, which is "
            << setw( 5 ) << ( 180/pi ) * va1 [ i ]
            << " degrees" << endl;
    cout << endl;

    va2 = sinh ( va1 );
    cout << "The hyperbolic sine of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

The initial valarray is:

-3.14159	radians, which is	-180	degrees
-2.35619	radians, which is	-135	degrees
-1.5708	radians, which is	-90	degrees
-0.785398	radians, which is	-45	degrees
0	radians, which is	0	degrees
0.785398	radians, which is	45	degrees
1.5708	radians, which is	90	degrees
2.35619	radians, which is	135	degrees
3.14159	radians, which is	180	degrees

The hyperbolic sine of the initial valarray is:

-11.5487
-5.22797
-2.3013
-0.868671
0
0.868671
2.3013
5.22797
11.5487

sqrt

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the square root of the elements of the input valarray.

```
template <class Type>
valarray<Type> sqrt(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the square root of the elements of the input valarray.

Example

```
// valarray_sqrt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <cmath>

int main( )
{
    using namespace std;
    int i;

    valarray<double> va1 ( 6 );
    for ( i = 0 ; i < 5 ; i++ )
        va1 [ i ] = i * i;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 5 ; i++ )
        cout << va1 [ i ] << " ";
    cout << ")." << endl;

    valarray<double> va2 = sqrt ( va1 );
    cout << "The square root of the initial valarray is: ( ";
    for ( i = 0 ; i < 5 ; i++ )
        cout << va2 [ i ] << " ";
    cout << ")." << endl;
}
```

```
The initial valarray is: ( 0 1 4 9 16 ).
The square root of the initial valarray is: ( 0 1 2 3 4 ).
```

swap

Exchanges the elements of two valarrays.

```
template <class Type>
void swap(
    valarray<Type>& left,
    valarray<Type>& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>left</i>	An object of type <code>valarray</code> .
<i>right</i>	An object of type <code>valarray</code> .

Remarks

The template function executes `left.swap(right)` .

tan

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the tangent of the elements of the input valarray.

```
template <class Type>
valarray<Type> tan(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the tangent of the elements of the input valarray.

Example

```
// valarray_tan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = ( pi/2 ) * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw( 10 ) << va1 [ i ]
        << "      radians, which is "
        << setw( 5 ) << ( 180/pi ) * va1 [ i ]
        << "      degrees" << endl;
    cout << endl;

    va2 = tan ( va1 );
    cout << "The tangent of the initial valarray is:\n ";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}
```

```
The initial valarray is:
-1.5708      radians, which is    -90    degrees
-1.1781      radians, which is   -67.5   degrees
-0.785398    radians, which is   -45    degrees
-0.392699    radians, which is   -22.5   degrees
      0      radians, which is      0    degrees
 0.392699    radians, which is    22.5   degrees
 0.785398    radians, which is    45    degrees
 1.1781      radians, which is    67.5   degrees
 1.5708      radians, which is    90    degrees
```

```
The tangent of the initial valarray is:
9.6701e+012
-2.41421
-1
-0.414214
0
0.414214
1
2.41421
-9.6701e+012
```

tanh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic tangent of the elements of the input valarray.

```
template <class Type>
valarray<Type> tanh(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic tangent in terms of the exponential function:

$$\tanh (z)=\sinh (z) / \cosh (z)=(\exp (z)-\exp (-z)) /(\exp (z)+\exp (-z))$$

Example

```

// valarray_tanh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;
    int i;

    valarray<double> va1 ( 9 );
    for ( i = 0 ; i < 9 ; i++ )
        va1 [ i ] = pi * ( 0.25 * i - 1 );
    valarray<double> va2 ( 9 );

    cout << "The initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << setw( 10 ) << va1 [ i ]
            << " radians, which is "
                << setw( 5 ) << ( 180/pi ) * va1 [ i ]
                << " degrees" << endl;
    cout << endl;

    va2 = tanh ( va1 );
    cout << "The hyperbolic tangent of the initial valarray is:\n";
    for ( i = 0 ; i < 9 ; i++ )
        cout << va2 [ i ] << endl;
}

```

```

The initial valarray is:
-3.14159 radians, which is -180 degrees
-2.35619 radians, which is -135 degrees
-1.5708 radians, which is -90 degrees
-0.785398 radians, which is -45 degrees
0 radians, which is 0 degrees
0.785398 radians, which is 45 degrees
1.5708 radians, which is 90 degrees
2.35619 radians, which is 135 degrees
3.14159 radians, which is 180 degrees

```

```

The hyperbolic tangent of the initial valarray is:
-0.996272
-0.982193
-0.917152
-0.655794
0
0.655794
0.917152
0.982193
0.996272

```

See also

[<valarray>](#)

<valarray> operators

3/28/2019 • 35 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator%</code>	<code>operator&</code>
<code>operator&&</code>	<code>operator></code>	<code>operator>></code>
<code>operator>=</code>	<code>operator<</code>	<code>operator<<</code>
<code>operator<=</code>	<code>operator*</code>	<code>operator+</code>
<code>operator-</code>	<code>operator/</code>	<code>operator==</code>
<code>operator^</code>	<code>operator </code>	<code>operator </code>

operator!=

Tests whether the corresponding elements of two equally sized valarrays are unequal or whether all the elements of a valarray are unequal a specified value.

```
template <class Type>
valarray<bool>
operator!=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator!=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator!=(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two valarrays whose elements are to be tested for inequality.

right

The second of the two valarrays whose elements are to be tested for inequality.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the corresponding elements are unequal.
- **false** if the corresponding elements are not unequal.

Remarks

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `left[I] != right[I]`.

The second template operator stores in element `I` `left[I] != right`.

The third template operator stores in element `I` `left != right[I]`.

Example

```
// valarray_op_ne.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL != vaR );
    cout << "The element-by-element result of "
        << "the not equal comparison test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the not equal comparison test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
*/
```

operator%

Obtains the remainder of dividing the corresponding elements of two equally sized valarrays or of dividing a valarray by a specified value or of dividing a specified value by a valarray.

```
template <class Type>
valarray<Type>
operator%(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator%(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator%(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

A value or valarray that serves as the dividend into which another value or valarray is to be divided.

right

A value or valarray that serves as the divisor and that divides another value or valarray.

Return Value

A valarray whose elements are the element-wise remainders of *left* divided by *right*.

Example

```

// valarray_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 6 ), vaR ( 6 );
    valarray<int> vaREM ( 6 );
    for ( i = 0 ; i < 6 ; i += 2 )
        vaL [ i ] = 53;
    for ( i = 1 ; i < 6 ; i += 2 )
        vaL [ i ] = -67;
    for ( i = 0 ; i < 6 ; i++ )
        vaR [ i ] = 3*i+1;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaREM = ( vaL % vaR );
    cout << "The remainders from the element-by-element "
        << "division is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaREM [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 53 -67 53 -67 53 -67 ).
The initial Right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
valarray: ( 0 -3 4 -7 1 -3 ).
*/

```

operator&

Obtains the bitwise **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```

template <class Type>
valarray<Type>
operator&(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator&(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator&(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose respective elements are to be combined with the bitwise `AND` or a specified value of the element type to be combined bitwise with each element of a valarray.

right

The second of the two valarrays whose respective elements are to be combined with the bitwise `AND` or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise AND operation of *left* and *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **longdouble**, **void**, **bool** or other, more complex data types.

The bitwise `AND` has the same truth table as the logical `AND` but applies to the data type on the level of the individual bits. The `operator&&` applies on an element level, counting all nonzero values as true, and the result is a valarray of Boolean values. The bitwise `AND` `operator&`, by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```

// valarray_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<int> vaBWA ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i+1;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaBWA = ( vaL & vaR );
    cout << "The element-by-element result of "
        << "the bitwise operator & is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaBWA [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 2 0 4 0 6 0 8 0 10 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the bitwise operator & is the
valarray: ( 0 0 0 0 0 4 0 0 0 8 ).
*/

```

operator&&

Obtains the logical **AND** between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray's element type.

```

template <class Type>
valarray<bool>
operator&&(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator&&(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator&&(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose respective elements are to be combined with the logical `AND` or a specified value of the element type to be combined with each element of a valarray.

right

The second of the two valarrays whose respective elements are to be combined with the logical `AND` or a specified value of the element type to be combined with each element of a valarray.

Return Value

A valarray whose elements are of type `bool` and are the element-wise combination of the logical `AND` operation of *left* and *right*.

Remarks

The logical `ANDoperator&&` applies on an element level, counting all nonzero values as true, and the result is a valarray of Boolean values. The bitwise version of `AND`, `operator&`, by contrast, can result in a valarray of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

```

// valarray_op_logand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaLAA ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i-1;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaLAA = ( vaL && vaR );
    cout << "The element-by-element result of "
        << "the logical AND operator&& is the\n"
        << "valarray: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaLAA [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the logical AND operator&& is the
valarray: ( 0 0 0 1 0 1 0 1 0 1 ).
*/

```

operator>

Tests whether the elements of one valarray are greater than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.

```

template <class Type>
valarray<bool>
operator>(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator>(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator>(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *left* element or value is greater than the corresponding *right* element or value.
- **false** if the *left* element or value is not greater than the corresponding *right* element or value.

Remarks

If the number of elements in two valarrays is not equal, the result is undefined.

Example


```

// valarray_op_gt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i - 1;

    cout << "The initial Left valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL > vaR );
    cout << "The element-by-element result of "
        << "the greater than comparison test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than comparison test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
*/

```

operator>=

Tests whether the elements of one valarray are greater than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.

```

template <class Type>
valarray<bool>
operator>=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator>=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator>=(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *left* element or value is greater than or equal to the corresponding *right* element or value.
- **false** if the *left* element or value is less than the corresponding *right* element or value.

Remarks

If the number of elements in two valarrays is not equal, the result is undefined.

Example

```

// valarray_op_ge.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i - 1;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL >= vaR );
    cout << "The element-by-element result of "
        << "the greater than or equal test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than or equal test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
*/

```

operator>>

Right-shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.

```
template <class Type>
valarray<Type>
operator>>(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator>>(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator>>(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The value to be shifted or the valarray whose elements are to be shifted.

right

The value indicating the amount of right shift or valarray whose elements indicate the element-wise amount of right shift.

Return Value

A valarray whose elements have been shifted right by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

```

// valarray_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    valarray<int> vaNE ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 64;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = -64;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL >> vaR );
    cout << "The element-by-element result of "
        << "the right shift is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
*/

```

operator<

Tests whether the elements of one valarray are less than the elements of an equally sized valarray or whether all the elements of a valarray are greater or less than a specified value.

```

template <class Type>
valarray<bool>
operator<(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator<(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator<(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *left* element or value is less than the corresponding *right* element or value.
- **false** if the *left* element or value is not less than the corresponding *right* element or value.

Remarks

If the number of elements two valarrays is not equal, the result is undefined.

Example

```

// valarray_op_lt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL < vaR );
    cout << "The element-by-element result of "
        << "the less-than comparison test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the less-than comparison test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
*/

```

operator<=

Tests whether the elements of one valarray are less than or equal to the elements of an equally sized valarray or whether all the elements of a valarray are greater than or equal to or less than or equal to a specified value.

```

template <class Type>
valarray<bool>
operator<=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator<=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator<=(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

right

The second of the two valarrays whose elements are to be compared or a specified value to be compared with each element of a valarray.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the *left* element or value is less than or equal to the corresponding *right* element or value.
- **false** if the *left* element or value is greater than the corresponding *right* element or value.

Remarks

If the number of elements two valarrays is not equal, the result is undefined.

Example


```

// valarray_op_le.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i - 1;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL <= vaR );
    cout << "The element-by-element result of "
        << "the less than or equal test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the less than or equal test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
*/

```

operator<<

Left shifts the bits for each element of a valarray a specified number of positions or by an element-wise amount specified by a second valarray.

```
template <class Type>
valarray<Type>
operator<<(  
    const valarray<Type>& left,  
    const valarray<Type>& right);  
  
template <class Type>  
valarray<Type>  
operator<<(  
    const valarray<Type>& left,  
    const Type& right);  
  
template <class Type>  
valarray<Type>  
operator<<(  
    const Type& left,  
    const valarray<Type>& right);
```

Parameters

left

The value to be shifted or the valarray whose elements are to be shifted.

right

The value indicating the amount of left shift or valarray whose elements indicate the element-wise amount of left shift.

Return Value

A valarray whose elements have been shifted left by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

```

// valarray_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    valarray<int> vaNE ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 1;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL << vaR );
    cout << "The element-by-element result of "
        << "the left shift is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift is the
valarray: ( 1 -2 4 -8 16 -32 64 -128 ).
*/

```

operator*

Obtains the element-wise product between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template <class Type>
valarray<Type>
operator*(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator*(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator*(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two valarrays whose elements are to be multiplied or a specified value to be multiplied with each element of a valarray.

right

The second of the two valarrays whose elements are to be multiplied or a specified value to be multiplied with each element of a valarray.

Return Value

A valarray whose elements are the element-wise product of *left* and *right*.

Example

```

// valarray_op_eprod.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    valarray<int> vaNE ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 2;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL * vaR );
    cout << "The element-by-element result of "
        << "the multiplication is the\n"
        << "valarray: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
*/

```

operator+

Obtains the element-wise sum between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template <class Type>
valarray<Type>
operator+(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator+(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator+(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two valarrays whose elements are to be added or a specified value to be added with each element of a valarray.

right

The second of the two valarrays whose elements are to be added or a specified value to be added with each element of a valarray.

Return Value

A valarray whose elements are the element-wise sum of *left* and *right*.

Example

```

// valarray_op_esum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    valarray<int> vaNE ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 2;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL + vaR );
    cout << "The element-by-element result of "
        << "the sum is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the sum is the
valarray: ( 2 0 4 2 6 4 8 6 ).
*/

```

operator-

Obtains the element-wise difference between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template <class Type>
valarray<Type>
operator-(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator-(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator-(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

A value or valarray that serves as the minuend from which other values or valarrays are to be subtracted in forming the difference.

right

A value or valarray that serves as the subtrahend that is to be subtracted from other values or valarrays in forming the difference.

Return Value

A valarray whose elements are the element-wise difference of *left* and *right*.

Remarks

The arithmetic terminology used in describing a subtraction:

difference = minuend - subtrahend

Example


```

// valarray_op_ediff.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    valarray<int> vaNE ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 10;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL - vaR );
    cout << "The element-by-element result of "
        << "the difference is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
*/

```

operator/

Obtains the element-wise quotient between corresponding elements of two equally sized valarrays or of between a valarray a specified value.

```
template <class Type>
valarray<Type>
operator/(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator/(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator/(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

A value or valarray that serves as the dividend into which another value or valarray is to be divided in forming the quotient.

right

A value or valarray that serves as the divisor and that divides another value or valarray in forming the quotient.

Return Value

A valarray whose elements are the element-wise quotient of *left* divided by *right*.

Remarks

The arithmetic terminology used in describing a division:

quotient = dividend / divisor

Example

```

// valarray_op_equo.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<double> vaL ( 6 ), vaR ( 6 );
    valarray<double> vaNE ( 6 );
    for ( i = 0 ; i < 6 ; i += 2 )
        vaL [ i ] = 100;
    for ( i = 1 ; i < 6 ; i += 2 )
        vaL [ i ] = -100;
    for ( i = 0 ; i < 6 ; i++ )
        vaR [ i ] = 2*i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL / vaR );
    cout << "The element-by-element result of "
        << "the quotient is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
valarray: ( inf -50 25 -16.6667 12.5 -10 ).
*/

```

operator==

Tests whether the corresponding elements of two equally sized valarrays are equal or whether all the elements of a valarray are equal a specified value.

```

template <class Type>
valarray<bool>
operator==(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator==(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator==(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose elements are to be tested for equality.

right

The second of the two valarrays whose elements are to be tested for equality.

Return Value

A valarray of Boolean values, each of which is:

- **true** if the corresponding elements are equal.
- **false** if the corresponding elements are not equal.

Remarks

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `left[I] == right[I]`. The second template operator stores in element `I` `left[I] == right`. The third template operator stores in element `I` `left == right[I]`.

Example

```

// valarray_op_eq.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaNE ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaNE = ( vaL == vaR );
    cout << "The element-by-element result of "
        << "the equality comparison test is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNE [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the equality comparison test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
*/

```

operator^

Obtains the bitwise exclusive OR (**XOR**) between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```

template <class Type>
valarray<Type>
operator^(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator^(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator^(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose respective elements are to be combined with the bitwise **XOR** or a specified value of the element type to be combined bitwise with each element of a valarray.

right

The second of the two valarrays whose respective elements are to be combined with the bitwise **XOR** or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise **XOR** operation of *left* and *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **long double**, **void**, **bool** or other, more complex data types.

The bitwise exclusive OR (**XOR**) has the following semantics: Given bits *b1* and *b2*, *b1 XOR b2* is **true** if exactly one of the bits is true; **false** if both bits are false or if both bits are true.

Example

```

// valarray_op_xor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<int> vaLAA ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 1;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaLAA = ( vaL ^ vaR );
    cout << "The element-by-element result of "
        << "the bitwise XOR operator^ is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaLAA [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^ is the
valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
*/

```

operator|

Obtains the bitwise OR between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the element type.

```

template <class Type>
valarray<Type>
operator|(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator|(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator|(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose respective elements are to be combined with the bitwise `OR` or a specified value of the element type to be combined bitwise with each element of a valarray.

right

The second of the two valarrays whose respective elements are to be combined with the bitwise `OR` or a specified value of the element type to be combined bitwise with each element of a valarray.

Return Value

A valarray whose elements are the element-wise combination of the bitwise `OR` operation of *left* and *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **longdouble**, **void**, **bool** or other, more complex data types.

The bitwise OR has the same truth table as the logical `OR`, but applies to the data type on the level of the individual bits. Given bits *b1* and *b2*, *b1 OR b2* is **true** if at least one of the bits is true or **false** if both bits are false. The logical `OR operator||` applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise OR `operator|`, by contrast, can result in a valarray of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example


```

// valarray_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<int> vaLAA ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 1;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;

    cout << "The initial Left valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaLAA = ( vaL | vaR );
    cout << "The element-by-element result of "
        << "the bitwise OR operator| is the\n"
        << "valarray: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << vaLAA [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise OR operator| is the
valarray: ( 1 0 1 3 3 4 7 6 7 9 ).
*/

```

operator||

Obtains the logical `OR` between corresponding elements of two equally sized valarrays or between a valarray and a specified value of the valarray element type.

```

template <class Type>
valarray<bool>
operator||(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator||(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator||(
    const Type& left,
    const valarray<Type>& right);

```

Parameters

left

The first of the two valarrays whose respective elements are to be combined with the logical `OR` or a specified value of the element type to be combined with each element of a valarray.

right

The second of the two valarrays whose respective elements are to be combined with the logical `OR` or a specified value of the element type to be combined with each element of a valarray.

Return Value

A valarray whose elements are of type **bool** and are the element-wise combination of the logical OR operation of *left* and *right*.

Remarks

The logical `OR` `operator||` applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise version of `OR`, `operator|` by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```

// valarray_op_logor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaLOR ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i-1;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = 0;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = 0;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaLOR = ( vaL || vaR );
    cout << "The element-by-element result of "
        << "the logical OR operator|| is the\n"
        << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaLOR [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 0 0 3 0 0 6 0 0 9 ).
The element-by-element result of the logical OR operator|| is the
valarray: ( 0 0 0 1 0 1 1 1 0 1 ).
*/

```

See also

[<valarray>](#)

gslice Class

10/31/2018 • 7 minutes to read • [Edit Online](#)

A utility class to valarray that is used to define multidimensional subsets of a valarray. If a valarray is regarded as a multidimensional matrix with all elements in an array, then the slice extracts a vector out of the multidimensional array.

Remarks

The class stores the parameters that characterize an object of type [gslice_array](#). The subset of a valarray is indirectly constructed when an object of class gslice appears as an argument for an object of class [valarray<Type>](#). The stored values that specify the subset selected from the parent valarray include:

- A starting index.
- A length vector of class `valarray<size_t>`.
- A stride vector of class `valarray<size_t>`.

The two vectors must have the same length.

If the set defined by a gslice is the subset of a constant valarray, then the gslice is a new valarray. If the set defined by a gslice is the subset of a nonconstant valarray, then the gslice has reference semantics to the original valarray. The evaluation mechanism for nonconstant valarrays saves time and memory.

Operations on valarrays are guaranteed only if the source and destination subsets defined by the gslices are distinct and all indices are valid.

Constructors

CONSTRUCTOR	DESCRIPTION
gslice	Defines a subset of a <code>valarray</code> that consists of multiple slices of the <code>valarray</code> that all start at a specified element.

Member functions

MEMBER FUNCTION	DESCRIPTION
size	Finds the array values specifying the numbers of elements in a general slice of a <code>valarray</code> .
start	Finds the starting index of a general slice of a <code>valarray</code> .
stride	Finds the distance between elements in a general slice of a <code>valarray</code> .

Requirements

Header: <valarray>

Namespace: std

gslice::gslice

A utility class to valarray that is used to define multi-dimensional slices of a valarray.

```
gslice();

gslice(
    size_t _StartIndex,
    const valarray<size_t>& _LenArray,
    const valarray<size_t>& _IncArray);
```

Parameters

_StartIndex

The valarray index of the first element in the subset.

_LenArray

An array specifying the number of elements in each slice.

_IncArray

An array specifying the stride in each slice.

Return Value

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores *_StartIndex* for the starting index, *_LenArray* for the length array, and *_IncArray* for the stride array.

Remarks

gslice defines a subset of a valarray that consists of multiple slices of the valarray that each start at the same specified element. The ability to use arrays to define multiple slices is the only difference between `gslice` and `slice::slice`. The first slice has a first element with an index of *_StartIndex*, a number of elements specified by the first element of *_LenArray*, and a stride given by the first element of *_IncArray*. The next set of orthogonal slices has first elements given by the first slice. The second element of *_LenArray* specifies the number of elements. The stride is given by the second element of *_IncArray*. A third dimension of slices would take the elements of the two-dimensional array as the starting elements and proceed analogously

Example

```

// gslice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i+=1 )
        va [ i ] = i;

    cout << "The operand valarray va is:" << endl << "(";
    for ( i = 0 ; i < 20 ; i++ )
        cout << " " << va [ i ];
    cout << " )" << endl;

    valarray<size_t> Len ( 2 ), Stride ( 2 );
    Len [0] = 4;
    Len [1] = 4;
    Stride [0] = 7;
    Stride [1] = 4;

    gslice vaGSlice ( 0, Len, Stride );
    vaResult = va [ vaGSlice ];

    cout << "The valarray for vaGSlice is vaResult:" << endl
        << "va[vaGSlice] = (";

    for ( i = 0 ; i < 8 ; i++ )
        cout << " " << vaResult [ i ];
    cout << ")" << endl;
}

```

```

The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 )
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19)

```

gslice::size

Finds the array values specifying the numbers of elements in a general slice of a valarray.

```
valarray<size_t> size() const;
```

Return Value

A valarray specifying the number of elements in each slice of a general slice of a valarray.

Remarks

The member function returns the stored lengths of slices.

Example

```

// gslice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    size_t sizeVA;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i+=1 )
        va [ i ] = i;

    cout << "The operand valarray va is:\n ( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    sizeVA = va.size ( );
    cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

    valarray<size_t> Len ( 2 ), Stride ( 2 );
    Len [0] = 4;
    Len [1] = 4;
    Stride [0] = 7;
    Stride [1] = 4;

    gslice vaGSlice ( 0, Len, Stride );
    vaResult = va [ vaGSlice ];
    const valarray <size_t> sizeGS = vaGSlice.size ( );

    cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    cout << "The size of vaResult is:"
        << "\n vaGSlice.size ( ) = ( ";
    for ( i = 0 ; i < 2 ; i++ )
        cout << sizeGS[ i ] << " ";
    cout << ")." << endl;
}

```

```

The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The size of the valarray is: 20.

The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The size of vaResult is:
vaGSlice.size ( ) = ( 4 4 ).

```

gslice::start

Finds the starting index of a general slice of a valarray.

```
size_t start() const;
```

Return Value

The starting index of a general slice of a valarray.

Example

```
// gslice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 20 ), vaResult;
    for (i = 0 ; i < 20 ; i+=1 )
        va [ i ] = i;

    cout << "The operand valarray va is:\n ( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    valarray<size_t> Len ( 2 ), Stride ( 2 );
    Len [0] = 4;
    Len [1] = 4;
    Stride [0] = 7;
    Stride [1] = 4;

    gslice vaGSlice ( 0, Len, Stride );
    vaResult = va [ vaGSlice ];
    size_t vaGSstart = vaGSlice.start ( );

    cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    cout << "The index of the first element of vaResult is: "
        << vaGSstart << "." << endl;
}
```

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The index of the first element of vaResult is: 0.
```

gslice::stride

Finds the distance between elements in a general slice of a valarray.

```
valarray<size_t> stride() const;
```

Return Value

A valarray specifying the distances between elements in each slice of a general slice of a valarray.

Example


```

// gslice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 20 ), vaResult;
    for (i = 0 ; i < 20 ; i+=1 )
        va [ i ] = i;

    cout << "The operand valarray va is:\n ( ";
        for (i = 0 ; i < 20 ; i++ )
            cout << va [ i ] << " ";
    cout << ")." << endl;

    valarray<size_t> Len ( 2 ), Stride ( 2 );
    Len [0] = 4;
    Len [1] = 4;
    Stride [0] = 7;
    Stride [1] = 4;

    gslice vaGSlice ( 0, Len, Stride );
    vaResult = va [ vaGSlice ];
    const valarray <size_t> strideGS = vaGSlice.stride ( );

    cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
        for ( i = 0 ; i < 8 ; i++ )
            cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    cout << "The strides of vaResult are:"
        << "\n vaGSlice.stride ( ) = ( ";
        for ( i = 0 ; i < 2 ; i++ )
            cout << strideGS[ i ] << " ";
    cout << ")." << endl;
}

```

```

The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The strides of vaResult are:
vaGSlice.stride ( ) = ( 7 4 ).

```

See also

[Thread Safety in the C++ Standard Library](#)

gslice_array Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

An internal, auxiliary template class that supports general slice objects by providing operations between subset arrays defined by the general slice of a valarray.

Syntax

```
template <class Type>
class gslice_array : public gsplce {
public:
    typedef Type value_type;
    void operator=(const valarray<Type>& x) const;

    void operator=(const Type& x) const;

    void operator*=(const valarray<Type>& x) const;

    void operator/=(const valarray<Type>& x) const;

    void operator%=(const valarray<Type>& x) const;

    void operator+=(const valarray<Type>& x) const;

    void operator-=(const valarray<Type>& x) const;

    void operator^=(const valarray<Type>& x) const;

    void operator&=(const valarray<Type>& x) const;

    void operator|=(const valarray<Type>& x) const;

    void operator<<=(const valarray<Type>& x) const;

    void operator>>=(const valarray<Type>& x) const;

    // The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object `va` of class `valarray<Type>`, along with an object `gs` of class `gslice` which describes the sequence of elements to select from the `valarray<Type>` object.

You construct a `gslice_array<Type>` object only by writing an expression of the form `va[gs]`. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The template class is created indirectly by certain `valarray` operations and cannot be used directly in the program. An internal auxiliary template class instead is used by the slice subscript operator:

```
gslice_array< Type> valarray< Type>::operator[] ( constgslice&).
```

You construct a `gslice_array<Type>` object only by writing an expression of the form `va[gs1]`, for a slice `gs1` of `valarray va`. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected. The sequence

controlled by the `gslice_array` is defined by the three parameters of the slice constructor, the index of the first element in the first slice, the number of elements in each slice, and the distance between the elements in each slice.

In the following example:

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);

// va[gslice(3, len, str)] selects elements with
//   indices 3, 5, 7, 10, 12, 14
```

The indices must be valid for the procedure to be valid.

Example

See the example for [gslice::gslice](#) for an example of how to declare and use a `slice_array`.

Requirements

Header: `<valarray>`

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

indirect_array Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

An internal, auxiliary template class that supports objects that are subsets of valarrays by providing operations between subset arrays defined by specifying a subset of indices of a parent valarray.

Syntax

Remarks

The class describes an object that stores a reference to an object `va` of class `valarray<Type>`, along with an object `xa` of class `valarray<size_t>`, which describes the sequence of elements to select from the `valarray<Type>` object.

You construct an `indirect_array<Type>` object only by writing an expression of the form `va[xa]`. The member functions of class `indirect_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The sequence consists of `xa.size` elements, where element `I` becomes the index `xa[I]` within `va`.

Example:

```
// indirect_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        va [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 2 )
        va [ i ] = -1;

    cout << "The initial operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    // Select 2nd, 4th & 6th elements
    // and assign a value of 10 to them
    valarray<size_t> indx ( 3 );
    indx [0] = 2;
    indx [1] = 4;
    indx [2] = 6;
    va[indx] = 10;

    cout << "The modified operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial operand valarray is: (0 -1 2 -1 4 -1 6 -1 8 -1).  
The modified operand valarray is: (0 -1 10 -1 10 -1 10 -1 8 -1).
```

Requirements

Header: <valarray>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

mask_array Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

An internal, auxiliary template class that supports objects that are subsets of parent valarrays, specified with a Boolean expression, by providing operations between the subset arrays.

Syntax

Remarks

The class describes an object that stores a reference to an object `va` of class `valarray<Type>`, along with an object `ba` of class `valarray<bool>`, which describes the sequence of elements to select from the `valarray<Type>` object.

You construct a `mask_array<Type>` object only by writing an expression of the form `va[ba]`. The member functions of class `mask_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The sequence consists of at most `ba.size` elements. An element J is included only if `ba[J]` is true. Thus, there are as many elements in the sequence as there are true elements in `ba`. If `I` is the index of the lowest true element in `ba`, then `va[I]` is element zero in the selected sequence.

Example

```
// mask_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        va [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 2 )
        va [ i ] = -1;

    cout << "The initial operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    // Use masked subsets to assign a value of 10
    // to all elements greater than 3 in value
    va [va > 3 ] = 10;
    cout << "The modified operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial operand valarray is: (0 -1 2 -1 4 -1 6 -1 8 -1).  
The modified operand valarray is: (0 -1 2 -1 10 -1 10 -1 10 -1).
```

Requirements

Header: <valarray>

Namespace: std

See also

[Thread Safety in the C++ Standard Library](#)

slice Class

10/31/2018 • 6 minutes to read • [Edit Online](#)

A utility class to valarray that is used to define one-dimensional subsets of a parent valarray. If a valarray is regarded as a two-dimensional matrix with all elements in an array, then the slice extracts a vector in one dimension out of the two-dimensional array.

Remarks

The class stores the parameters that characterize an object of type `slice_array`. The subset of a valarray is indirectly constructed when an object of class slice appears as an argument for an object of class `valarray<Type>`. The stored values that specify the subset selected from the parent valarray include:

- A starting index in the valarray.
- A total length, or number of elements in the slice.
- A stride, or distance between subsequent indices of elements in the valarray.

If the set defined by a slice is the subset of a constant valarray, then the slice is a new valarray. If the set defined by a slice is the subset of a nonconstant valarray, then the slice has reference semantics to the original valarray. The evaluation mechanism for nonconstant valarrays saves time and memory.

Operations on valarrays are guaranteed only if the source and destination subsets defined by the slices are distinct and all indices are valid.

Constructors

CONSTRUCTOR	DESCRIPTION
<code>slice</code>	Defines a subset of a <code>valarray</code> that consists of a number of elements that are an equal distance apart and that start at a specified element.

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>size</code>	Finds the number of elements in a slice of a <code>valarray</code> .
<code>start</code>	Finds the starting index of a slice of a <code>valarray</code> .
<code>stride</code>	Finds the distance between elements in a slice of a <code>valarray</code> .

Requirements

Header: `<valarray>`

Namespace: `std`

`slice::size`

Finds the number of elements in a slice of a valarray.

```
size_t size() const;
```

Return Value

The number of elements in a slice of a valarray.

Example

```
// slice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    size_t sizeVA, sizeVAR;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i += 1 )
        va [ i ] = i+1;

    cout << "The operand valarray va is:\n ( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    sizeVA = va.size ( );
    cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

    slice vaSlice ( 3 , 6 , 3 );
    vaResult = va [ vaSlice ];

    cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    sizeVAR = vaSlice.size ( );
    cout << "The size of slice vaSlice is: "
        << sizeVAR << "." << endl;
}
```

```
The operand valarray va is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The size of the valarray is: 20.

The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
( 4 7 10 13 16 19 ).
The size of slice vaSlice is: 6.
```

slice::slice

Defines a subset of a valarray that consists of a number of elements that are an equal distance apart and that start at a specified element.

```

slice();

slice(
    size_t _StartIndex,
    size_t _Len,
    size_t stride);

```

Parameters

_StartIndex

The valarray index of the first element in the subset.

_Len

The number of elements in the subset.

stride

The distance between elements in the subset.

Return Value

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores *_StartIndex* for the starting index, *_Len* for the total length, and *stride* for the stride.

Remarks

The stride may be negative.

Example

```

// slice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i+=1 )
        va [ i ] = 2 * ( i + 1 );

    cout << "The operand valarray va is:\n( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    slice vaSlice ( 1 , 7 , 3 );
    vaResult = va [ vaSlice ];

    cout << "\nThe slice of valarray va is vaResult:"
        << "\nva[slice( 1, 7, 3)] = ( ";
    for ( i = 0 ; i < 7 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;
}

```

The operand valarray va is:
(2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40).

The slice of valarray va is vaResult:
va[slice(1, 7, 3)] = (4 10 16 22 28 34 40).

slice::start

Finds the starting index of a slice of a valarray.

```
size_t start() const;
```

Return Value

The starting index of a slice of a valarray.

Example

```
// slice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    size_t startVAR;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i += 1 )
        va [ i ] = i+1;

    cout << "The operand valarray va is:\n ( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    slice vaSlice ( 3 , 6 , 3 );
    vaResult = va [ vaSlice ];

    cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    startVAR = vaSlice.start ( );
    cout << "The start index of slice vaSlice is: "
        << startVAR << "." << endl;
}
```

```
The operand valarray va is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
( 4 7 10 13 16 19 ).
The start index of slice vaSlice is: 3.
```

slice::stride

Finds the distance between elements in a slice of a valarray.

```
size_t stride() const;
```

Return Value

The distance between elements in a slice of a valarray.

Example

```
// slice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    size_t strideVAR;

    valarray<int> va ( 20 ), vaResult;
    for ( i = 0 ; i < 20 ; i += 1 )
        va [ i ] = 3 * ( i + 1 );

    cout << "The operand valarray va is:\n ( ";
    for ( i = 0 ; i < 20 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    slice vaSlice ( 4 , 5 , 3 );
    vaResult = va [ vaSlice ];

    cout << "The slice of valarray va is vaResult = "
        << "va[slice( 4, 5, 3)] =\n ( ";
    for ( i = 0 ; i < 5 ; i++ )
        cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    strideVAR = vaSlice.stride ( );
    cout << "The stride of slice vaSlice is: "
        << strideVAR << "." << endl;
}
```

```
The operand valarray va is:
( 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 ).
The slice of valarray va is vaResult = va[slice( 4, 5, 3)] =
( 15 24 33 42 51 ).
The stride of slice vaSlice is: 3.
```

See also

[Thread Safety in the C++ Standard Library](#)

slice_array Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

An internal, auxiliary template class that supports slice objects by providing operations between subset arrays defined by the slice of a valarray.

Syntax

```
template <class Type>
class slice_array : public slice {
public:
    typedef Type value_type;
    void operator=(const valarray<Type>& x) const;
    void operator=(const Type& x) const;
    void operator*=(const valarray<Type>& x) const;
    void operator/=(const valarray<Type>& x) const;
    void operator%=(const valarray<Type>& x) const;
    void operator+=(const valarray<Type>& x) const;
    void operator-=(const valarray<Type>& x) const;
    void operator^=(const valarray<Type>& x) const;
    void operator&=(const valarray<Type>& x) const;
    void operator|=(const valarray<Type>& x) const;
    void operator<=(const valarray<Type>& x) const;
    void operator>=(const valarray<Type>& x) const;
    // The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object of class `valarray<Type>`, along with an object of class `slice`, which describes the sequence of elements to select from the `valarray<Type>` object.

The template class is created indirectly by certain valarray operations and cannot be used directly in the program. An internal, auxiliary template class that is used by the slice subscript operator:

```
slice_array < Type> valarray < Type>::operator[] ( slice ).
```

You construct a `slice_array<Type>` object only by writing an expression of the form `va[s]`, for a slice `s` of valarray `va`. The member functions of class `slice_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected. The sequence controlled by the `slice_array` is defined by the three parameters of the slice constructor, the index of the first element in the slice, the number of elements, and the distance between the elements. A `slice_array` cut from valarray `va` declared by `va[slice(2, 5, 3)]` selects elements with indices 2, 5, 8, 11, and 14 from `va`. The indices must be valid for the procedure to be valid.

Example

See the example for `slice::slice` for an example of how to declare and use a `slice_array`.

Requirements

Header: <valarray>

Namespace: `std`

See also

[Thread Safety in the C++ Standard Library](#)

valarray Class

3/28/2019 • 43 minutes to read • [Edit Online](#)

The template class describes an object that controls a sequence of elements of type `Type` that are stored as an array, designed for performing high-speed mathematical operations, and optimized for computational performance.

Remarks

The class is a representation of the mathematical concept of an ordered set of values and the elements are numbered sequentially from zero. The class is described as a near container because it supports some, but not all, of the capabilities that first-class sequence containers, such as [vector](#), support. It differs from template class `vector` in two important ways:

- It defines numerous arithmetic operations between corresponding elements of `valarray<Type>` objects of the same type and length, such as $xarr = \cos(yarr) + \sin(zarr)$.
- It defines a variety of interesting ways to subscript a `valarray<Type>` object, by overloading `operator[]`.

An object of class `Type`:

- Has a public default constructor, destructor, copy constructor, and assignment operator, with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types, with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class `Type` may throw exceptions.

Constructors

CONSTRUCTOR	DESCRIPTION
valarray	Constructs a <code>valarray</code> of a specific size or with elements of a specific value or as a copy of another <code>valarray</code> or subset of another <code>valarray</code> .

Typedefs

TYPE NAME	DESCRIPTION
value_type	A type that represents the type of element stored in a <code>valarray</code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
apply	Applies a specified function to each element of a <code>valarray</code> .

MEMBER FUNCTION	DESCRIPTION
<code>cshift</code>	Cyclically shifts all the elements in a <code>valarray</code> by a specified number of positions.
<code>free</code>	Frees the memory used by the <code>valarray</code> .
<code>max</code>	Finds the largest element in a <code>valarray</code> .
<code>min</code>	Finds the smallest element in a <code>valarray</code> .
<code>resize</code>	Changes the number of elements in a <code>valarray</code> to a specified number, adding or removing elements as required.
<code>shift</code>	Shifts all the elements in a <code>valarray</code> by a specified number of positions.
<code>size</code>	Finds the number of elements in a <code>valarray</code> .
<code>sum</code>	Determines the sum of all the elements in a <code>valarray</code> of nonzero length.
<code>swap</code>	

Operators

OPERATOR	DESCRIPTION
<code>operator!</code>	A unary operator that obtains the logical <code>NOT</code> values of each element in a <code>valarray</code> .
<code>operator% =</code>	Obtains the remainder of dividing the elements of an array element-wise either by a specified <code>valarray</code> or by a value of the element type.
<code>operator& =</code>	Obtains the bitwise <code>AND</code> of elements in an array either with the corresponding elements in a specified <code>valarray</code> or with a value of the element type.
<code>operator>> =</code>	Right-shifts the bits for each element of a <code>valarray</code> operand a specified number of positions or by an element-wise amount specified by a second <code>valarray</code> .
<code>operator<< =</code>	Left-shifts the bits for each element of a <code>valarray</code> operand a specified number of positions or by an element-wise amount specified by a second <code>valarray</code> .
<code>operator* =</code>	Multiplies the elements of a specified <code>valarray</code> or a value of the element type, element-wise, to an operand <code>valarray</code> .
<code>operator+</code>	A unary operator that applies a plus to each element in a <code>valarray</code> .

OPERATOR	DESCRIPTION
<code>operator+=</code>	Adds the elements of a specified <code>valarray</code> or a value of the element type, element-wise, to an operand <code>valarray</code> .
<code>operator-</code>	A unary operator that applies a minus to each element in a <code>valarray</code> .
<code>operator-=</code>	Subtracts the elements of a specified <code>valarray</code> or a value of the element type, element-wise, from an operand <code>valarray</code> .
<code>operator/=</code>	Divides an operand <code>valarray</code> element-wise by the elements of a specified <code>valarray</code> or a value of the element type.
<code>operator=</code>	Assigns elements to a <code>valarray</code> whose values are specified either directly or as part of some other <code>valarray</code> or by a <code>slice_array</code> , <code>gslice_array</code> , <code>mask_array</code> , or <code>indirect_array</code> .
<code>operator[]</code>	Returns a reference to an element or its value at specified index or a specified subset.
<code>operator^=</code>	Obtains the element-wise exclusive logical or operator (<code>XOR</code>) of an array with either a specified <code>valarray</code> or a value of the element type.
<code>operator =</code>	Obtains the bitwise <code>OR</code> of elements in an array either with the corresponding elements in a specified <code>valarray</code> or with a value of the element type.
<code>operator~</code>	A unary operator that obtains the bitwise <code>NOT</code> values of each element in a <code>valarray</code> .

Requirements

Header: `<valarray>`

Namespace: `std`

`valarray::apply`

Applies a specified function to each element of a `valarray`.

```
valarray<Type> apply(Type _Func(Type)) const;

valarray<Type> apply(Type _Func(constType&)) const;
```

Parameters

`_Func(Type)`

The function object to be applied to each element of the operand `valarray`.

`_Func(const Type&)`

The function object for `const` to be applied to each element of the operand `valarray`.

Return Value

A valarray whose elements have had `_Func` applied element-wise to the elements of the operand valarray.

Remarks

The member function returns an object of class `valarray<Type>`, of length `size`, each of whose elements `i` is `_Func((*this)[i])`.

Example

```
// valarray_apply.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

using namespace std;

int __cdecl MyApplyFunc( int n )
{
    return n*2;
}

int main( int argc, char* argv[] )
{
    valarray<int> vaR(10), vaApplied(10);
    int i;

    for ( i = 0; i < 10; i += 3 )
        vaR[i] = i;

    for ( i = 1; i < 10; i += 3 )
        vaR[i] = 0;

    for ( i = 2; i < 10; i += 3 )
        vaR[i] = -i;

    cout << "The initial Right valarray is: (" ;
    for ( i=0; i < 10; ++i )
        cout << " " << vaR[i];
    cout << " )" << endl;

    vaApplied = vaR.apply( MyApplyFunc );

    cout << "The element-by-element result of "
        << "applying MyApplyFunc to vaR is the\nvalarray: ( ";
    for ( i = 0; i < 10; ++i )
        cout << " " << vaApplied[i];
    cout << " )" << endl;
}
/* Output:
The initial Right valarray is: ( 0 0 -2 3 0 -5 6 0 -8 9 )
The element-by-element result of applying MyApplyFunc to vaR is the
valarray: ( 0 0 -4 6 0 -10 12 0 -16 18 )
*/
```

valarray::cshift

Cyclically shifts all the elements in a valarray by a specified number of positions.

```
valarray<Type> cshift(int count) const;
```

Parameters

count

The number of places the elements are to be shifted forward.

Return Value

A new valarray in which all the elements have been moved *count* positions cyclically toward the front of the valarray, left with respect to their positions in the operand valarray.

Remarks

A positive value of *count* shifts the elements cyclically left *count* places.

A negative value of *count* shifts the elements cyclically right *count* places.

Example

```
// valarray_cshift.cpp
// compile with: /EHsc

#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;

    valarray<int> va1(10), va2(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;
    for (i = 0; i < 10; i+=1)
        va2[i] = 10 - i;

    cout << "The operand valarray va1 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    // A positive parameter shifts elements right
    va1 = va1.cshift(4);
    cout << "The cyclically shifted valarray va1 is:\nva1.cshift (4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    cout << "The operand valarray va2 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;

    // A negative parameter shifts elements left
    va2 = va2.cshift(-4);
    cout << "The cyclically shifted valarray va2 is:\nva2.shift (-4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;
}

/* Output:
The operand valarray va1 is: ( 0 1 2 3 4 5 6 7 8 9)
The cyclically shifted valarray va1 is:
va1.cshift (4) = ( 4 5 6 7 8 9 0 1 2 3)
The operand valarray va2 is: ( 10 9 8 7 6 5 4 3 2 1)
The cyclically shifted valarray va2 is:
va2.shift (-4) = ( 4 3 2 1 10 9 8 7 6 5)
*/
```

valarray::free

Frees the memory used by the valarray.

```
void free();
```

Remarks

This nonstandard function is equivalent to assigning an empty valarray. For example:

```
valarray<T> v;  
v = valarray<T>();  
  
// equivalent to v.free()
```

valarray::max

Finds the largest element in a valarray.

```
Type max() const;
```

Return Value

The maximum value of the elements in the operand valarray.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class `Type`, for which operators must be provided for the element `Type`.

Example

```
// valarray_max.cpp  
// compile with: /EHsc  
#include <valarray>  
#include <iostream>  
  
int main( )  
{  
    using namespace std;  
    int i, MaxValue;  
  
    valarray<int> vaR ( 10 );  
    for ( i = 0 ; i < 10 ; i += 3 )  
        vaR [ i ] = i;  
    for ( i = 1 ; i < 10 ; i += 3 )  
        vaR [ i ] = 2*i - 1;  
    for ( i = 2 ; i < 10 ; i += 3 )  
        vaR [ i ] = 10 - i;  
  
    cout << "The operand valarray is: ( ";  
    for ( i = 0 ; i < 10 ; i++ )  
        cout << vaR [ i ] << " ";  
    cout << ")." << endl;  
  
    MaxValue = vaR.max ( );  
    cout << "The largest element in the valarray is: "  
        << MaxValue << "." << endl;  
}  
/* Output:  
The operand valarray is: ( 0 1 8 3 7 5 6 13 2 9 ).  
The largest element in the valarray is: 13.  
*/
```

valarray::min

Finds the smallest element in a valarray.

```
Type min() const;
```

Return Value

The minimum value of the elements in the operand valarray.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class `Type`, for which operators must be provided for the element `Type`.

Example

```
// valarray_min.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i, MinValue;

    valarray<int> vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = 2*i;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = 5 - i;

    cout << "The operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    MinValue = vaR.min ( );
    cout << "The smallest element in the valarray is: "
        << MinValue << "." << endl;
}
/* Output:
The operand valarray is: ( 0 2 3 -3 8 0 -6 14 -3 -9 ).
The smallest element in the valarray is: -9.
*/
```

valarray::operator!

A unary operator that obtains the logical **NOT** values of each element in a valarray.

```
valarray<bool> operator!() const;
```

Return Value

The valarray of Boolean values that are the negation of the element values of the operand valarray.

Remarks

The logical operation **NOT** negates the elements because it converts all zeros into ones and regards all nonzero

values as ones and converts them into zeros. The returned valarray of Boolean values is of the same size as the operand valarray.

There is also a bitwise **NOT** `valarray::operator~` that negates on the level of individual bits within the binary representation of **char** and **int** elements of a valarray.

Example

```
// valarray_op_lognot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 10 );
    valarray<bool> vaNOT ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val [ i ] = 0;
    for ( i = 1 ; i < 10 ; i += 2 )
        val [ i ] = i-1;

    cout << "The initial valarray is: ( ";
    for (i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    vaNOT = !val;
    cout << "The element-by-element result of "
        << "the logical NOT operator! is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNOT [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 0 0 0 2 0 4 0 6 0 8 ).
The element-by-element result of the logical NOT operator! is the
valarray: ( 1 1 1 0 1 0 1 0 1 0 ).
*/
```

valarray::operator%=

Obtains the remainder of dividing the elements of an array element-wise either by a specified valarray or by a value of the element type.

```
valarray<Type>& operator%=(const valarray<Type>& right);

valarray<Type>& operator%=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to divide, element-wise, the operand valarray.

Return Value

A valarray whose elements are the remainder from the element-wise division of the operand valarray by *right*

Example

```
// valarray_class_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 6 ), vaR ( 6 );
    for ( i = 0 ; i < 6 ; i += 2 )
        val [ i ] = 53;
    for ( i = 1 ; i < 6 ; i += 2 )
        val [ i ] = -67;
    for ( i = 0 ; i < 6 ; i++ )
        vaR [ i ] = 3*i+1;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial right valarray is: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val %= vaR;
    cout << "The remainders from the element-by-element "
        << "division is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 6 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 53 -67 53 -67 53 -67 ).
The initial right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
valarray: ( 0 -3 4 -7 1 -3 ).
*/
```

valarray::operator&=

Obtains the bitwise **AND** of elements in an array either with the corresponding elements in a specified valarray or with a value of the element type.

```
valarray<Type>& operator&=(const valarray<Type>& right);

valarray<Type>& operator&=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the logical **AND** with the operand valarray.

Return Value

A valarray whose elements are the element-wise logical **AND** of the operand valarray by *right*

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **longdouble**, **void**, **bool**, or other, more complex data types.

The bitwise AND has the same truth table as the logical `AND` but applies to the data type on the level of the individual bits. Given bits $b1$ and $b2$, $b1$ `AND` $b2$ is **true** if both bits are true; **false** if at least one is false.

Example

```
// valarray_class_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 10 ), vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val [ i ] = 0;
    for ( i = 1 ; i < 10 ; i += 2 )
        val [ i ] = i-1;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val &= vaR;
    cout << "The element-by-element result of "
        << "the logical AND operator&= is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the logical AND operator&= is the
valarray: ( 0 0 0 2 0 4 0 6 0 8 ).
*/
```

valarray::operator>>=

Right-shifts the bits for each element of a valarray operand a specified number of positions or by an element-wise amount specified by a second valarray.

```
valarray<Type>& operator>>=(const valarray<Type>& right);

valarray<Type>& operator>>=(const Type& right);
```

Parameters

right

The value indicating the amount of right shift or valarray whose elements indicate the element-wise amount of right shift.

Return Value

A valarray whose elements have been shifted right the amount specified in *right*.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_class_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 8 ), vaR ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        vaL [ i ] = 64;
    for ( i = 1 ; i < 8 ; i += 2 )
        vaL [ i ] = -64;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial operand valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaL >>= vaR;
    cout << "The element-by-element result of "
        << "the right shift is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial operand valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
*/
```

valarray::operator<<=

Left-shifts the bits for each element of a valarray operand a specified number of positions or by an element-wise amount specified by a second valarray.

```
valarray<Type>& operator<<=(const valarray<Type>& right);
```

```
valarray<Type>& operator<<=(const Type& right);
```

Parameters

right

The value indicating the amount of left shift or valarray whose elements indicate the element-wise amount of left shift.

Return Value

A valarray whose elements have been shifted left the amount specified in *right*.

Remarks

Signed numbers have their signs preserved.

Example

```
// valarray_class_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 8 ), vaR ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        val [ i ] = 1;
    for ( i = 1 ; i < 8 ; i += 2 )
        val [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial operand valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val <<= vaR;
    cout << "The element-by-element result of "
        << "the left shift"
        << endl << "on the operand array is the valarray:"
        << endl << "( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial operand valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift
on the operand array is the valarray:
( 1 -2 4 -8 16 -32 64 -128 ).
*/
```

valarray::operator*=

Multiplies the elements of a specified valarray or a value of the element type, element-wise, to an operand valarray.

```
valarray<Type>& operator*=(const valarray<Type>& right);

valarray<Type>& operator*=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to multiply, element-wise, the operand valarray.

Return Value

A valarray whose elements are the element-wise product of the operand valarray and *right*.

Example

```
// valarray_op_emult.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 8 ), vaR ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        val [ i ] = 2;
    for ( i = 1 ; i < 8 ; i += 2 )
        val [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val *= vaR;
    cout << "The element-by-element result of "
        << "the multiplication is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
*/
```

valarray::operator+

A unary operator that applies a plus to each element in a valarray.

```
valarray<Type> operator+() const;
```

Return Value

A valarray whose elements are plus those of the operand array.

Example

```
// valarray_op_eplus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 10 );
    valarray<int> vaPLUS ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        val [ i ] = i-1;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    vaPLUS = +val;
    cout << "The element-by-element result of "
        << "the operator+ is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaPLUS [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator+ is the
valarray: ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
*/
```

valarray::operator+=

Adds the elements of a specified valarray or a value of the element type, element-wise, to an operand valarray.

```
valarray<Type>& operator+=(const valarray<Type>& right);

valarray<Type>& operator+=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be added, element-wise, to the operand valarray.

Return Value

A valarray whose elements are the element-wise sum of the operand valarray and *right*.

Example

```
// valarray_op_eadd.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 8 ), vaR ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        val [ i ] = 2;
    for ( i = 1 ; i < 8 ; i += 2 )
        val [ i ] = -1;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial valarray is: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial  right valarray is: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val += vaR;
    cout << "The element-by-element result of "
        << "the sum is the"
        << endl << "valarray: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial  right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the sum is the
valarray: ( 2 0 4 2 6 4 8 6 ).
*/
```

valarray::operator-

A unary operator that applies a minus to each element in a valarray.

```
valarray<Type> operator-() const;
```

Return Value

A valarray whose elements are minus those of the operand array.

Example

```

// valarray_op_eminus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 10 );
    valarray<int> vaMINUS ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        val [ i ] = i-1;

    cout << "The initial valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    vaMINUS = -val;
    cout << "The element-by-element result of "
        << "the operator+ is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaMINUS [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator+ is the
valarray: ( 0 0 2 -2 4 -4 6 -6 8 -8 ).
*/

```

valarray::operator-=

Subtracts the elements of a specified valarray or a value of the element type, element-wise, from an operand valarray.

```

valarray<Type>& operator-=(const valarray<Type>& right);

valarray<Type>& operator-=(const Type& right);

```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be subtracted, element-wise, from the operand valarray.

Return Value

A valarray whose elements are the element-wise difference of the operand valarray and *right*.

Example

```

// valarray_op_esub.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 8 ), vaR ( 8 );
    for ( i = 0 ; i < 8 ; i += 2 )
        val [ i ] = 10;
    for ( i = 1 ; i < 8 ; i += 2 )
        val [ i ] = 0;
    for ( i = 0 ; i < 8 ; i++ )
        vaR [ i ] = i;

    cout << "The initial valarray is: ( ";
    for (i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial  right valarray is: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val -= vaR;
    cout << "The element-by-element result of "
        << "the difference is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 8 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial  right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
*/

```

valarray::operator/=

Divides an operand valarray element-wise by the elements of a specified valarray or a value of the element type.

```

valarray<Type>& operator/=(const valarray<Type>& right);

valarray<Type>& operator/=(const Type& right);

```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be divided, element-wise, into the operand valarray.

Return Value

A valarray whose elements are the element-wise quotient of the operand valarray divided by *right*.

Example

```

// valarray_op_ediv.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<double> val ( 6 ), vaR ( 6 );
    for ( i = 0 ; i < 6 ; i += 2 )
        val [ i ] = 100;
    for ( i = 1 ; i < 6 ; i += 2 )
        val [ i ] = -100;
    for ( i = 0 ; i < 6 ; i++ )
        vaR [ i ] = 2*i;

    cout << "The initial valarray is: ( ";
    for (i = 0 ; i < 6 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for (i = 0 ; i < 6 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val /= vaR;
    cout << "The element-by-element result of "
        << "the quotient is the"
        << endl << "valarray: ( ";
    for (i = 0 ; i < 6 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
valarray: ( inf -50 25 -16.6667 12.5 -10 ).
*/

```

valarray::operator=

Assigns elements to a valarray whose values are specified either directly or as part of some other valarray or by a slice_array, gslice_array, mask_array, or indirect_array.

```

valarray<Type>& operator=(const valarray<Type>& right);

valarray<Type>& operator=(valarray<Type>&& right);

valarray<Type>& operator=(const Type& val);

valarray<Type>& operator=(const slice_array<Type>& _Slicearray);

valarray<Type>& operator=(const gslice_array<Type>& _Gslicearray);

valarray<Type>& operator=(const mask_array<Type>& _Maskarray);

valarray<Type>& operator=(const indirect_array<Type>& _Indarray);

```

Parameters

right

The valarray to be copied into the operand valarray.

val

The value to be assigned to the elements of the operand valarray.

_Slicearray

The slice_array to be copied into the operand valarray.

_Gslicearray

The gslice_array to be copied into the operand valarray.

_Maskarray

The mask_array to be copied into the operand valarray.

_Indarray

The indirect_array to be copied into the operand valarray.

Return Value

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *right*.

The second member operator is the same as the first, but with an [Rvalue Reference Declarator: &&](#).

The third member operator replaces each element of the controlled sequence with a copy of *val*.

The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by [operator\[\]](#).

If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

Remarks

If the length of the controlled sequence changes, the result is generally undefined. In this implementation, however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

Example

```

// valarray_op_assign.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va ( 10 ), vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 1 )
        va [ i ] = i;
    for ( i = 0 ; i < 10 ; i+=1 )
        vaR [ i ] = 10 - i;

    cout << "The operand valarray va is:";
    for ( i = 0 ; i < 10 ; i++ )
        cout << " " << va [ i ];
    cout << endl;

    cout << "The operand valarray vaR is:";
    for ( i = 0 ; i < 10 ; i++ )
        cout << " " << vaR [ i ];
    cout << endl;

    // Assigning vaR to va with the first member function
    va = vaR;
    cout << "The reassigned valarray va is:";
    for ( i = 0 ; i < 10 ; i++ )
        cout << " " << va [ i ];
    cout << endl;

    // Assigning elements of value 10 to va
    // with the second member function
    va = 10;
    cout << "The reassigned valarray va is:";
    for ( i = 0 ; i < 10 ; i++ )
        cout << " " << va [ i ];
    cout << endl;
}
/* Output:
The operand valarray va is: 0 1 2 3 4 5 6 7 8 9
The operand valarray vaR is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 10 10 10 10 10 10 10 10 10
*/

```

valarray::operator[]

Returns a reference to an element or its value at specified index or a specified subset.

```

Type& operator[](size_t _Off);

slice_array<Type> operator[](slice _Slicearray);

gslice_array<Type> operator[](const gslice& _Gslicearray);

mask_array<Type> operator[](const valarray<bool>& _Boolarray);

indirect_array<Type> operator[](const valarray<size_t>& _Indarray);

Type operator[](size_t _Off) const;

valarray<Type> operator[](slice _Slice) const;

valarray<Type> operator[](const gslice& _Gslicearray) const;

valarray<Type> operator[](const valarray<bool>& _Boolarray) const;

valarray<Type> operator[](const valarray<size_t>& _Indarray) const;

```

Parameters

_Off

The index of the element to be assigned a value.

_Slicearray

A slice_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Gslicearray

A gslice_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Boolarray

A bool_array of a valarray that specifies a subset to be selected or returned to a new valarray.

_Indarray

An indirect_array of a valarray that specifies a subset to be selected or returned to a new valarray.

Return Value

A reference to an element or its value at specified index or a specified subset.

Remarks

The member operator is overloaded to provide several ways to select sequences of elements from among those controlled by ***this**. The first group of five member operators work in conjunction with various overloads of [operator=](#) (and other assigning operators) to allow selective replacement (slicing) of the controlled sequence. The selected elements must exist.

When compiled by using [_ITERATOR_DEBUG_LEVEL](#) defined as 1 or 2, a runtime error occurs if you attempt to access an element outside the bounds of the valarray. See [Checked Iterators](#) for more information.

Example

See the examples for [slice::slice](#) and [gslice::gslice](#) for an example of how to declare and use the operator.

valarray::operator^=

Obtains the element-wise exclusive logical or operator (**XOR**) of an array with either a specified valarray or a value of the element type.

```
valarray<Type>& operator|=(const valarray<Type>& right);
```

```
valarray<Type>& operator|=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the exclusive logical **XOR** with the operand valarray.

Return Value

A valarray whose elements are the element-wise, exclusive logical **XOR** of the operand valarray and *right*.

Remarks

The exclusive logical or, referred to as **XOR**, has the following semantics: Given elements *e1* and *e2*, *e1 XOR e2* is **true** if exactly one of the elements is true; **false** if both elements are false or if both elements are true.

Example

```

// valarray_op_exor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = 1;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = 0;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;

    cout << "The initial operand valarray is: ( ";
        for (i = 0 ; i < 10 ; i++ )
            cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The  right valarray is: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaL ^= vaR;
    cout << "The element-by-element result of "
        << "the bitwise XOR operator^= is the"
        << endl << "valarray: ( ";
        for (i = 0 ; i < 10 ; i++ )
            cout << vaL [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial operand valarray is: ( 1 0 1 0 1 0 1 0 1 0 ).
The  right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^= is the
valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
*/

```

valarray::operator|=

Obtains the bitwise **OR** of elements in an array either with the corresponding elements in a specified valarray or with a value of the element type.

```

valarray<Type>& operator|=(const valarray<Type>& right);

valarray<Type>& operator|=(const Type& right);

```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be combined, element-wise, by the bitwise **OR** with the operand valarray.

Return Value

A valarray whose elements are the element-wise bitwise `OR` of the operand valarray by *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **longdouble**, **void**, **bool**, or other, more complex data types.

The bitwise `OR` has the same truth table as the logical `OR` but applies to the data type on the level of the individual bits. Given bits *b1* and *b2*, *b1 OR b2* is **true** if at least one of the bits is true; **false** if both bits are false.

Example

```
// valarray_class_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> val ( 10 ), vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val [ i ] = 1;
    for ( i = 1 ; i < 10 ; i += 2 )
        val [ i ] = 0;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] = i-1;

    cout << "The initial operand valarray is:"
         << endl << "( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;

    cout << "The right valarray is:"
         << endl << "( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    val |= vaR;
    cout << "The element-by-element result of "
         << "the logical OR"
         << endl << "operator|= is the valarray:"
         << endl << "( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial operand valarray is:
( 1 0 1 0 1 0 1 0 1 0 ).
The right valarray is:
( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the logical OR
operator|= is the valarray:
( 1 0 1 3 3 4 7 6 7 9 ).
*/
```

valarray::operator~

A unary operator that obtains the bitwise `NOT` values of each element in a valarray.

```
valarray<Type> operator~() const;
```

Return Value

The valarray of Boolean values that are the bitwise `NOT` of the element values of the operand valarray.

Remarks

A bitwise operation can only be used to manipulate bits in **char** and **int** data types and variants and not on **float**, **double**, **longdouble**, **void**, **bool** or other, more complex data types.

The bitwise `NOT` has the same truth table as the logical `NOT` but applies to the data type on the level of the individual bits. Given bit b , $\sim b$ is true if b is false and false if b is true. The logical **NOToperator!** applies on an element level, counting all nonzero values as **true**, and the result is a valarray of Boolean values. The bitwise `NOToperator~`, by contrast, can result in a valarray of values other than 0 or 1, depending on outcome of the bitwise operation.

Example

```
// valarray_op_bitnot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<unsigned short int> val1 ( 10 );
    valarray<unsigned short int> vaNOT1 ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val1 [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 2 )
        val1 [ i ] = 5*i;

    cout << "The initial valarray <unsigned short int> is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val1 [ i ] << " ";
    cout << ")." << endl;

    vaNOT1 = ~val1;
    cout << "The element-by-element result of "
        << "the bitwise NOT operator~ is the"
        << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNOT1 [ i ] << " ";
    cout << ")." << endl << endl;

    valarray<int> val2 ( 10 );
    valarray<int> vaNOT2 ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        val2 [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 2 )
        val2 [ i ] = -2 * i;

    cout << "The initial valarray <int> is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << val2 [ i ] << " ";
    cout << ")." << endl;
```

```

vaNOT2 = ~vaL2;
cout << "The element-by-element result of "
    << "the bitwise NOT operator~ is the"
    << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNOT2 [ i ] << " ";
cout << ")." << endl;

// The negative numbers are represented using
// the two's complement approach, so adding one
// to the flipped bits returns the negative elements
vaNOT2 = vaNOT2 + 1;
cout << "The element-by-element result of "
    << "adding one"
    << endl << "is the negative of the "
    << "original elements the"
    << endl << "valarray: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaNOT2 [ i ] << " ";
cout << ")." << endl;
}

/* Output:
The initial valarray <unsigned short int> is: ( 0 5 2 15 4 25 6 35 8 45 ).
The element-by-element result of the bitwise NOT operator~ is the
valarray: ( 65535 65530 65533 65520 65531 65510 65529 65500 65527 65490 ).

The initial valarray <int> is: ( 0 -2 2 -6 4 -10 6 -14 8 -18 ).
The element-by-element result of the bitwise NOT operator~ is the
valarray: ( -1 1 -3 5 -5 9 -7 13 -9 17 ).
The element-by-element result of adding one
is the negative of the original elements the
valarray: ( 0 2 -2 6 -4 10 -6 14 -8 18 ).
*/

```

valarray::resize

Changes the number of elements in a valarray to a specified number.

```

void resize(
    size_t _Newsize);

void resize(
    size_t _Newsize,
    const Type val);

```

Parameters

_Newsize

The number of elements in the resized valarray.

val

The value to be given to the elements of the resized valarray.

Remarks

The first member function initializes elements with their default constructor.

Any pointers or references to elements in the controlled sequence are invalidated.

Example

The following example demonstrates the use of the valarray::resize member function.


```
// valarray_resize.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, sizeNew;

    valarray<int> va1(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;

    cout << "The valarray contains ( ";
    for (i = 0; i < 10; i++)
        cout << va1[i] << " ";
    cout << ")." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray is: "
        << size1 << "." << endl << endl;

    va1.resize(15, 10);

    cout << "The valarray contains ( ";
    for (i = 0; i < 15; i++)
        cout << va1[i] << " ";
    cout << ")." << endl;
    sizeNew = va1.size();
    cout << "The number of elements in the resized valarray is: "
        << sizeNew << "." << endl << endl;
}
```

```
The valarray contains ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray is: 10.
```

```
The valarray contains ( 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 ).
The number of elements in the resized valarray is: 15.
```

valarray::shift

Shifts all the elements in a valarray by a specified number of places.

```
valarray<Type> shift(int count) const;
```

Parameters

count

The number of places the elements are to be shifted forward.

Return Value

A new valarray in which all the elements have been moved *count* positions toward the front of the valarray, left with respect to their positions in the operand valarray.

Remarks

A positive value of *count* shifts the elements left *count* places, with zero fill.

A negative value of *count* shifts the elements right *count* places, with zero fill.

Example

```
// valarray_shift.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> va1 ( 10 ), va2 ( 10 );
    for ( i = 0 ; i < 10 ; i += 1 )
        va1 [ i ] = i;
    for ( i = 0 ; i < 10 ; i += 1 )
        va2 [ i ] = 10 - i;

    cout << "The operand valarray va1(10) is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va1 [ i ] << " ";
    cout << ")." << endl;

    // A positive parameter shifts elements left
    va1 = va1.shift ( 4 );
    cout << "The shifted valarray va1 is: va1.shift (4) = ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va1 [ i ] << " ";
    cout << ")." << endl;

    cout << "The operand valarray va2(10) is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va2 [ i ] << " ";
    cout << ")." << endl;

    // A negative parameter shifts elements right
    va2 = va2.shift ( - 4 );
    cout << "The shifted valarray va2 is: va2.shift (-4) = ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va2 [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The shifted valarray va1 is: va1.shift (4) = ( 4 5 6 7 8 9 0 0 0 0 ).
The operand valarray va2(10) is: ( 10 9 8 7 6 5 4 3 2 1 ).
The shifted valarray va2 is: va2.shift (-4) = ( 0 0 0 0 10 9 8 7 6 5 ).
*/
```

valarray::size

Finds the number of elements in a valarray.

```
size_t size() const;
```

Return Value

The number of elements in the operand valarray.

Example

The following example demonstrates the use of the valarray::size member function.

```

// valarray_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, size2;

    valarray<int> va1(10), va2(12);
    for (i = 0; i < 10; i += 1)
        va1[i] = i;
    for (i = 0; i < 10; i += 1)
        va2[i] = i;

    cout << "The operand valarray va1(10) is: ( ";
    for (i = 0; i < 10; i++)
        cout << va1[i] << " ";
    cout << ")." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray va1 is: va1.size = "
        << size1 << "." << endl << endl;

    cout << "The operand valarray va2(12) is: ( ";
    for (i = 0; i < 10; i++)
        cout << va2[i] << " ";
    cout << ")." << endl;

    size2 = va2.size();
    cout << "The number of elements in the valarray va2 is: va2.size = "
        << size2 << "." << endl << endl;

    // Initializing two more elements to va2
    va2[10] = 10;
    va2[11] = 11;
    cout << "After initializing two more elements,\n"
        << "the operand valarray va2(12) is now: ( ";
    for (i = 0; i < 12; i++)
        cout << va2[i] << " ";
    cout << ")." << endl;
    cout << "The number of elements in the valarray va2 is still: "
        << size2 << "." << endl;
}

```

```

The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va1 is: va1.size = 10.

```

```

The operand valarray va2(12) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va2 is: va2.size = 12.

```

```

After initializing two more elements,
the operand valarray va2(12) is now: ( 0 1 2 3 4 5 6 7 8 9 10 11 ).
The number of elements in the valarray va2 is still: 12.

```

valarray::sum

Determines the sum of all the elements in a valarray of nonzero length.

```
Type sum() const;
```

Return Value

The sum of the elements of the operand valarray.

Remarks

If the length is greater than one, the member function adds values to the sum by applying `operator+=` between pairs of elements of class `Type`, which operator, consequently, needs be provided for elements of type `Type`.

Example

```
// valarray_sum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    int sumva = 0;

    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i+=1 )
        va [ i ] = i;

    cout << "The operand valarray va (10) is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    sumva = va.sum ( );
    cout << "The sum of elements in the valarray is: "
        << sumva << "." << endl;
}
/* Output:
The operand valarray va (10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The sum of elements in the valarray is: 45.
*/
```

valarray::swap

Exchanges the elements of two `valarray`s.

```
void swap(valarray& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	A <code>valarray</code> providing the elements to be swapped.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. It does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences.

valarray::valarray

Constructs a valarray of a specific size or with elements of a specific value or as a copy of another valarray or

subset of another valarray.

```
valarray();

explicit valarray(
    size_t Count);

valarray(
    const Type& Val,
    size_t Count);

valarray(
    const Type* Ptr,
    size_t Count);

valarray(
    const valarray<Type>& Right);

valarray(
    const slice_array<Type>& SliceArray);

valarray(
    const gslice_array<Type>& GsliceArray);

valarray(
    const mask_array<Type>& MaskArray);

valarray(
    const indirect_array<Type>& IndArray);

valarray(
    valarray<Type>&& Right);

valarray(
    initializer_list<Type> IList);
```

Parameters

Count

The number of elements to be in the valarray.

Val

The value to be used in initializing the elements in the valarray.

Ptr

Pointer to the values to be used to initialize the elements in the valarray.

Right

An existing valarray to initialize the new valarray.

SliceArray

A slice_array whose element values are to be used in initializing the elements of the valarray being constructed.

GsliceArray

A gslice_array whose element values are to be used in initializing the elements of the valarray being constructed.

MaskArray

A mask_array whose element values are to be used in initializing the elements of the valarray being constructed.

IndArray

An indirect_array whose element values are to be used in initializing the elements of the valarray being constructed.

IList

The `initializer_list` containing the elements to copy.

Remarks

The first (default) constructor initializes the object to an empty array. The next three constructors each initialize the object to an array of *Count* elements as follows:

- For explicit `valarray(size_t Count)`, each element is initialized with the default constructor.
- For `valarray(const Type& Val, Count)`, each element is initialized with *Val*.
- For `valarray(const Type* Ptr, Count)`, the element at position *I* is initialized with `Ptr[I]`.

Each remaining constructor initializes the object to a `valarray<Type>` object determined by the subset specified in the argument.

The last constructor is the same as the next to last, but with an [Rvalue Reference Declarator](#): `&&`.

Example

```
// valarray_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;

    // The second member function
    valarray<int> va(10);
    for (auto i : va){
        va[i] = 2 * (i + 1);
    }

    cout << "The operand valarray va is:\n";
    for (auto i : va) {
        cout << " " << va[i];
    }
    cout << " )" << endl;

    slice Slice(2, 4, 3);

    // The fifth member function
    valarray<int> vaSlice = va[Slice];

    cout << "The new valarray initialized from the slice is vaSlice =\n"
        << "va[slice( 2, 4, 3)] = (" ;
    for (int i = 0; i < 3; i++) {
        cout << " " << vaSlice[i];
    }
    cout << " )" << endl;

    valarray<int> va2{{ 1, 2, 3, 4 }};
    for (auto& v : va2) {
        cout << v << " ";
    }
    cout << endl;
}
```

```
The operand valarray va is:
( 0 2 2 2 2 2 2 2 2 )
The new valarray initialized from the slice is vaSlice =
va[slice( 2, 4, 3)] = ( 0 0 0 )
1 2 3 4
```

valarray::value_type

A type that represents the type of element stored in a valarray.

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.

Example

```
// valarray_value_type.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        va [ i ] = i;
    for ( i = 1 ; i < 10 ; i += 2 )
        va [ i ] = -1;

    cout << "The initial operand valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;

    // value_type declaration and initialization:
    valarray<int>::value_type Right = 10;

    cout << "The decalared value_type Right is: "
        << Right << endl;
    va *= Right;
    cout << "The resulting valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << va [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial operand valarray is: ( 0 -1 2 -1 4 -1 6 -1 8 -1 ).
The decalared value_type Right is: 10
The resulting valarray is: ( 0 -10 20 -10 40 -10 60 -10 80 -10 ).
*/
```

See also

[Thread Safety in the C++ Standard Library](#)

valarray<bool> Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

A specialized version of the template class **valarray<Type>** to elements of type **bool**.

Syntax

```
class valarray<bool>
```

Example

```
// valarray_bool.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    valarray<bool> vaBool ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] = -i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] = i;
    for ( i = 0 ; i < 10 ; i++ )
        vaR [ i ] = i;

    cout << "The initial Left valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The initial Right valarray is: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaBool = ( vaL < vaR );
    cout << "The result of the less-than comparison "
    << "test is the\nvalarray<bool>: ( ";
    for ( i = 0 ; i < 10 ; i++ )
        cout << vaBool [ i ] << " ";
    cout << ")." << endl;
}

/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The result of the less-than comparison test is the
valarray<bool>: ( 0 0 1 0 1 0 1 0 1 0 ).
*/
```

Requirements

Header: <valarray>

Namespace: std

See also

[valarray Class](#)

[Thread Safety in the C++ Standard Library](#)

<vector>

10/31/2018 • 2 minutes to read • [Edit Online](#)

Defines the container template class `vector` and several supporting templates.

The `vector` is a container that organizes elements of a given type in a linear sequence. It enables fast random access to any element, and dynamic additions and removals to and from the sequence. The `vector` is the preferred container for a sequence when random-access performance is at a premium.

For more information about the class `vector`, see [vector Class](#). For information about the specialization `vector<bool>`, see [vector<bool> Class](#).

Syntax

```

namespace std {
template <class Type, class Allocator>
class vector;
template <class Allocator>
class vector<bool>;

template <class Allocator>
struct hash<vector<bool, Allocator>>;

// TEMPLATE FUNCTIONS
template <class Type, class Allocator>
bool operator== (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator!= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator< (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator> (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator<= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator>= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
void swap (
    vector<Type, Allocator>& left,
    vector<Type, Allocator>& right);

} // namespace std

```

Parameters

Type

The template parameter for the type of data stored in the vector.

Allocator

The template parameter for the stored allocator object responsible for memory allocation and deallocation.

left

The first (left) vector in a compare operation

right

The second (right) vector in a compare operation.

Operators

OPERATOR	DESCRIPTION
<code>operator!=</code>	Tests if the vector object on the left side of the operator is not equal to the vector object on the right side.
<code>operator<</code>	Tests if the vector object on the left side of the operator is less than the vector object on the right side.
<code>operator<=</code>	Tests if the vector object on the left side of the operator is less than or equal to the vector object on the right side.
<code>operator==</code>	Tests if the vector object on the left side of the operator is equal to the vector object on the right side.
<code>operator></code>	Tests if the vector object on the left side of the operator is greater than the vector object on the right side.
<code>operator>=</code>	Tests if the vector object on the left side of the operator is greater than or equal to the vector object on the right side.

Classes

CLASS	DESCRIPTION
<code>vector</code> Class	A template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element.

Specializations

<code>vector<bool></code> Class	A full specialization of the template class <code>vector</code> for elements of type <code>bool</code> with an allocator for the underlying type used by the specialization.
---------------------------------------	--

Requirements

Header: `<vector>`

Namespace: `std`

See also

[Header Files Reference](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

vector Class

12/14/2018 • 36 minutes to read • [Edit Online](#)

The C++ Standard Library vector class is a template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element. They should be the preferred container for a sequence when random-access performance is at a premium.

Syntax

```
template <class Type, class Allocator = allocator<Type>>
class vector
```

Parameters

Type

The element data type to be stored in the vector

Allocator

The type that represents the stored allocator object that encapsulates details about the vector's allocation and deallocation of memory. This argument is optional and the default value is `allocator<Type>`.

Remarks

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The performance of the [deque Class](#) container is superior with respect to insertions and deletions at the beginning and end of a sequence. The [list Class](#) container is superior with respect to insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The [vector<bool> Class](#) is a full specialization of the template class vector for elements of type bool with an allocator for the underlying type used by the specialization.

The [vector<bool> reference Class](#) is a nested class whose objects are able to provide references to elements (single bits) within a vector<bool> object.

Members

Constructors

CONSTRUCTOR	DESCRIPTION
vector	Constructs a vector of a specific size or with elements of a specific value or with a specific <code>allocator</code> or as a copy of some other vector.

Typedefs

TYPE NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the <code>allocator</code> class for the vector object.
<code>const_iterator</code>	A type that provides a random-access iterator that can read a const element in a vector.
<code>const_pointer</code>	A type that provides a pointer to a const element in a vector.
<code>const_reference</code>	A type that provides a reference to a const element stored in a vector for reading and performing const operations.
<code>const_reverse_iterator</code>	A type that provides a random-access iterator that can read any const element in the vector.
<code>difference_type</code>	A type that provides the difference between the addresses of two elements in a vector.
<code>iterator</code>	A type that provides a random-access iterator that can read or modify any element in a vector.
<code>pointer</code>	A type that provides a pointer to an element in a vector.
<code>reference</code>	A type that provides a reference to an element stored in a vector.
<code>reverse_iterator</code>	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
<code>size_type</code>	A type that counts the number of elements in a vector.
<code>value_type</code>	A type that represents the data type stored in a vector.

Member Functions

MEMBER FUNCTION	DESCRIPTION
<code>assign</code>	Erases a vector and copies the specified elements to the empty vector.
<code>at</code>	Returns a reference to the element at a specified location in the vector.
<code>back</code>	Returns a reference to the last element of the vector.
<code>begin</code>	Returns a random-access iterator to the first element in the vector.
<code>capacity</code>	Returns the number of elements that the vector could contain without allocating more storage.
<code>cbegin</code>	Returns a random-access const iterator to the first element in the vector.

MEMBER FUNCTION	DESCRIPTION
<code>cend</code>	Returns a random-access const iterator that points just beyond the end of the vector.
<code>crbegin</code>	Returns a const iterator to the first element in a reversed vector.
<code>crend</code>	Returns a const iterator to the end of a reversed vector.
<code>clear</code>	Erases the elements of the vector.
<code>data</code>	Returns a pointer to the first element in the vector.
<code>emplace</code>	Inserts an element constructed in place into the vector at a specified position.
<code>emplace_back</code>	Adds an element constructed in place to the end of the vector.
<code>empty</code>	Tests if the vector container is empty.
<code>end</code>	Returns a random-access iterator that points to the end of the vector.
<code>erase</code>	Removes an element or a range of elements in a vector from specified positions.
<code>front</code>	Returns a reference to the first element in a vector.
<code>get_allocator</code>	Returns an object to the <code>allocator</code> class used by a vector.
<code>insert</code>	Inserts an element or a number of elements into the vector at a specified position.
<code>max_size</code>	Returns the maximum length of the vector.
<code>pop_back</code>	Deletes the element at the end of the vector.
<code>push_back</code>	Add an element to the end of the vector.
<code>rbegin</code>	Returns an iterator to the first element in a reversed vector.
<code>rend</code>	Returns an iterator to the end of a reversed vector.
<code>reserve</code>	Reserves a minimum length of storage for a vector object.
<code>resize</code>	Specifies a new size for a vector.
<code>shrink_to_fit</code>	Discards excess capacity.
<code>size</code>	Returns the number of elements in the vector.

MEMBER FUNCTION	DESCRIPTION
swap	Exchanges the elements of two vectors.

Operators

OPERATOR	DESCRIPTION
operator[]	Returns a reference to the vector element at a specified position.
operator=	Replaces the elements of the vector with a copy of another vector.

Requirements

Header: <vector>

Namespace: std

vector::allocator_type

A type that represents the allocator class for the vector object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

Example

See the example for [get_allocator](#) for an example that uses `allocator_type`.

vector::assign

Erases a vector and copies the specified elements to the empty vector.

```
void assign(size_type Count, const Type& Val);
void assign(initializer_list<Type> Ilist);

template <class InputIterator>
void assign(InputIterator First, InputIterator Last);
```

Parameters

First

Position of the first element in the range of elements to be copied.

Last

Position of the first element beyond the range of elements to be copied.

Count

The number of copies of an element being inserted into the vector.

Val

The value of the element being inserted into the vector.

//List

The initializer_list containing the elements to insert.

Remarks

After erasing any existing elements in a vector, assign either inserts a specified range of elements from the original vector into a vector or inserts copies of a new element of a specified value into a vector.

Example

```
/ vector_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2, v3;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1 = ";
    for (auto& v : v1){
        cout << v << " ";
    }
    cout << endl;

    v2.assign(v1.begin(), v1.end());
    cout << "v2 = ";
    for (auto& v : v2){
        cout << v << " ";
    }
    cout << endl;

    v3.assign(7, 4);
    cout << "v3 = ";
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;

    v3.assign({ 5, 6, 7 });
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;
}
```

vector::at

Returns a reference to the element at a specified location in the vector.

```
reference at(size_type _Pos);

const_reference at(size_type _Pos) const;
```

Parameters

_Pos

The subscript or position number of the element to reference in the vector.

Return Value

A reference to the element subscripted in the argument. If `_off` is greater than the size of the vector, `at` throws an exception.

Remarks

If the return value of `at` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `at` is assigned to a `reference`, the vector object can be modified.

Example

```
// vector_at.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector <int> v1;

    v1.push_back( 10 );
    v1.push_back( 20 );

    const int &i = v1.at( 0 );
    int &j = v1.at( 1 );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;
}
```

```
The first element is 10
The second element is 20
```

vector::back

Returns a reference to the last element of the vector.

```
reference back();

const_reference back() const;
```

Return Value

The last element of the vector. If the vector is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `back` is assigned to a `reference`, the vector object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error occurs if you attempt to access an element in an empty vector. See [Checked Iterators](#) for more information.

Example

```

// vector_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;

    v1.push_back( 10 );
    v1.push_back( 11 );

    int& i = v1.back( );
    const int& ii = v1.front( );

    cout << "The last integer of v1 is " << i << endl;
    i--;
    cout << "The next-to-last integer of v1 is " << ii << endl;
}

```

vector::begin

Returns a random-access iterator to the first element in the vector.

```

const_iterator begin() const;

iterator begin();

```

Return Value

A random-access iterator addressing the first element in the `vector` or to the location succeeding an empty `vector`. You should always compare the value returned with `vector::end` to ensure it is valid.

Remarks

If the return value of `begin` is assigned to a `vector::const_iterator`, the `vector` object cannot be modified. If the return value of `begin` is assigned to an `vector::iterator`, the `vector` object can be modified.

Example

```

// vector_begin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::iterator c1_Iter;
    vector<int>::const_iterator c1_cIter;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_Iter = c1.begin();
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_Iter = c1.begin();
    *c1_Iter = 20;
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}

```

```

The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2

```

vector::capacity

Returns the number of elements that the vector could contain without allocating more storage.

```
size_type capacity() const;
```

Return Value

The current length of storage allocated for the vector.

Remarks

The member function [resize](#) will be more efficient if sufficient memory is allocated to accommodate it. Use the member function [reserve](#) to specify the amount of memory allocated.

Example

```
// vector_capacity.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector <int> v1;

    v1.push_back( 1 );
    cout << "The length of storage allocated is "
         << v1.capacity( ) << "." << endl;

    v1.push_back( 2 );
    cout << "The length of storage allocated is now "
         << v1.capacity( ) << "." << endl;
}
```

```
The length of storage allocated is 1.
The length of storage allocated is now 2.
```

vector::cbegin

Returns a **const** iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A **const** random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the [auto](#) type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `begin()` and `cbegin()`.

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

vector::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A **const** random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the `auto` type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-**const**) container of any kind that supports `end()` and `cend()`.

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

vector::clear

Erases the elements of the vector.

```
void clear();
```

Example

```
// vector_clear.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );

    cout << "The size of v1 is " << v1.size( ) << endl;
    v1.clear( );
    cout << "The size of v1 after clearing is " << v1.size( ) << endl;
}
```

```
The size of v1 is 3
The size of v1 after clearing is 0
```

vector::const_iterator

A type that provides a random-access iterator that can read a **const** element in a vector.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for [back](#) for an example that uses `const_iterator`.

vector::const_pointer

A type that provides a pointer to a **const** element in a vector.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

An [iterator](#) is more commonly used to access a vector element.

vector::const_reference

A type that provides a reference to a **const** element stored in a vector for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

Example

```
// vector_const_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;

    v1.push_back( 10 );
    v1.push_back( 20 );

    const vector<int> v2 = v1;
    const int &i = v2.front( );
    const int &j = v2.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

    // The following line would cause an error as v2 is const
    // v2.push_back( 30 );
}
```

```
The first element is 10
The second element is 20
```

vector::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the vector.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the vector in reverse.

Example

See [rbegin](#) for an example of how to declare and use an iterator.

vector::crbegin

Returns a const iterator to the first element in a reversed vector.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse random-access iterator addressing the first element in a reversed [vector](#) or addressing what had been the last element in the unreversed `vector`.

Remarks

With the return value of `crbegin`, the `vector` object cannot be modified.

Example

```
// vector_crbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator v1_Iter;
    vector<int>::const_reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    v1_Iter = v1.begin( );
    cout << "The first element of vector is "
         << *v1_Iter << "." << endl;

    v1_rIter = v1.crbegin( );
    cout << "The first element of the reversed vector is "
         << *v1_rIter << "." << endl;
}
```

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

vector::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed vector.


```
const_reverse_iterator crend() const;
```

Return Value

A const reverse random-access iterator that addresses the location succeeding the last element in a reversed **vector** (the location that had preceded the first element in the unreversed **vector**).

Remarks

crend is used with a reversed **vector** just as **vector::cend** is used with a **vector**.

With the return value of **crend** (suitably decremented), the **vector** object cannot be modified.

crend can be used to test to whether a reverse iterator has reached the end of its **vector**.

The value returned by **crend** should not be dereferenced.

Example

```
// vector_crend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::const_reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
        cout << *v1_rIter << endl;
}
```

```
2
1
```

vector::data

Returns a pointer to the first element in the vector.

```
const_pointer data() const;

pointer data();
```

Return Value

A pointer to the first element in the **vector** or to the location succeeding an empty **vector**.

Example

```

// vector_data.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::pointer c1_ptr;
    vector<int>::const_pointer c1_cPtr;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_cPtr = c1.data();
    for (size_t n = c1.size(); 0 < n; --n, c1_cPtr++)
    {
        cout << " " << *c1_cPtr;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_ptr = c1.data();
    *c1_ptr = 20;
    for (size_t n = c1.size(); 0 < n; --n, c1_ptr++)
    {
        cout << " " << *c1_ptr;
    }
    cout << endl;
}

```

```

The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2

```

vector::difference_type

A type that provides the difference between two iterators that refer to elements within the same vector.

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

A `difference_type` can also be described as the number of elements between two pointers, because a pointer to an element contains its address.

An [iterator](#) is more commonly used to access a vector element.

Example

```

// vector_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <vector>
#include <algorithm>

int main( )
{
    using namespace std;

    vector<int> c1;
    vector<int>::iterator c1_Iter, c2_Iter;

    c1.push_back( 30 );
    c1.push_back( 20 );
    c1.push_back( 30 );
    c1.push_back( 10 );
    c1.push_back( 30 );
    c1.push_back( 20 );

    c1_Iter = c1.begin( );
    c2_Iter = c1.end( );

    vector<int>::difference_type df_typ1, df_typ2, df_typ3;

    df_typ1 = count( c1_Iter, c2_Iter, 10 );
    df_typ2 = count( c1_Iter, c2_Iter, 20 );
    df_typ3 = count( c1_Iter, c2_Iter, 30 );
    cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
    cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
    cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}

```

```

The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.

```

vector::emplace

Inserts an element constructed in place into the vector at a specified position.

```

iterator emplace(
    const_iterator _Where,
    Type&& val);

```

Parameters

PARAMETER	DESCRIPTION
<i>_Where</i>	The position in the vector where the first element is inserted.
<i>val</i>	The value of the element being inserted into the <code>vector</code> .

Return Value

The function returns an iterator that points to the position where the new element was inserted into the `vector`.

Remarks

Any insertion operation can be expensive, see [vector Class](#) for a discussion of `vector` performance.

Example

```
// vector_emplace.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );

    cout << "v1 =" ;
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    // initialize a vector of vectors by moving v1
    vector< vector<int> > vv1;

    vv1.emplace( vv1.begin(), move( v1 ) );
    if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
    {
        cout << "vv1[0] =";
        for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
            cout << " " << *Iter;
        cout << endl;
    }
}
```

```
v1 = 10 20 30
vv1[0] = 10 20 30
```

vector::emplace_back

Adds an element constructed in place to the end of the vector.

```
template <class... Types>
void emplace_back(Types&&... _Args);
```

Parameters

PARAMETER	DESCRIPTION
<i>_Args</i>	Constructor arguments. The function infers which constructor overload to invoke based on the arguments provided.

Example

```
#include <vector>
struct obj
{
    obj(int, double) {}
};

int main()
{
    std::vector<obj> v;
    v.emplace_back(1, 3.14); // obj is created in place in the vector
}
```

vector::empty

Tests if the vector is empty.

```
bool empty() const;
```

Return Value

true if the vector is empty; **false** if the vector is not empty.

Example

```
// vector_empty.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;

    v1.push_back( 10 );

    if ( v1.empty( ) )
        cout << "The vector is empty." << endl;
    else
        cout << "The vector is not empty." << endl;
}
```

```
The vector is not empty.
```

vector::end

Returns the past-the-end iterator.

```
iterator end();

const_iterator end() const;
```

Return Value

The past-the-end iterator for the vector. If the vector is empty, `vector::end() == vector::begin()`.

Remarks

If the return value of `end` is assigned to a variable of type `const_iterator`, the vector object cannot be modified. If the return value of `end` is assigned to a variable of type `iterator`, the vector object can be modified.

Example

```
// vector_end.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
int main( )
{
    using namespace std;
    vector <int> v1;
    vector <int>::iterator v1_Iter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
        cout << *v1_Iter << endl;
}
```

```
1
2
```

vector::erase

Removes an element or a range of elements in a vector from specified positions.

```
iterator erase(
    const_iterator _Where);

iterator erase(
    const_iterator first,
    const_iterator last);
```

Parameters

PARAMETER	DESCRIPTION
<i>_Where</i>	Position of the element to be removed from the vector.
<i>first</i>	Position of the first element removed from the vector.
<i>last</i>	Position just beyond the last element removed from the vector.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists.

Example

```

// vector_erase.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );
    v1.push_back( 40 );
    v1.push_back( 50 );

    cout << "v1 =" ;
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    v1.erase( v1.begin( ) );
    cout << "v1 =";
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    v1.erase( v1.begin( ) + 1, v1.begin( ) + 3 );
    cout << "v1 =";
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;
}

```

```

v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50

```

vector::front

Returns a reference to the first element in a vector.

```

reference front();

const_reference front() const;

```

Return Value

A reference to the first element in the vector object. If the vector is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `front` is assigned to a **reference**, the vector object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error occurs if you attempt to access an element in an empty vector. See [Checked Iterators](#) for more information.

Example

```

// vector_front.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;

    v1.push_back( 10 );
    v1.push_back( 11 );

    int& i = v1.front( );
    const int& ii = v1.front( );

    cout << "The first integer of v1 is "<< i << endl;
    // by incrementing i, we move the front reference to the second element
    i++;
    cout << "Now, the first integer of v1 is "<< i << endl;
}

```

vector::get_allocator

Returns a copy of the allocator object used to construct the vector.

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the vector.

Remarks

Allocators for the vector class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```

// vector_get_allocator.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    // The following lines declare objects that use the default allocator.
    vector<int> v1;
    vector<int, allocator<int> > v2 = vector<int, allocator<int> >(allocator<int>( ) ) ;

    // v3 will use the same allocator class as v1
    vector<int> v3( v1.get_allocator( ) );

    vector<int>::allocator_type xvec = v3.get_allocator( );
    // You can now call functions on the allocator class used by vec
}

```

vector::insert

Inserts an element or a number of elements or a range of elements into the vector at a specified position.

```
iterator insert(
    const_iterator _Where,
    const Type& val);

iterator insert(
    const_iterator _Where,
    Type&& val);

void insert(
    const_iterator _Where,
    size_type count,
    const Type& val);

template <class InputIterator>
void insert(
    const_iterator _Where,
    InputIterator first,
    InputIterator last);
```

Parameters

PARAMETER	DESCRIPTION
<i>_Where</i>	The position in the vector where the first element is inserted.
<i>val</i>	The value of the element being inserted into the vector.
<i>count</i>	The number of elements being inserted into the vector.
<i>first</i>	The position of the first element in the range of elements to be copied.
<i>last</i>	The position of the first element beyond the range of elements to be copied.

Return Value

The first two `insert` functions return an iterator that points to the position where the new element was inserted into the vector.

Remarks

As a precondition, *first* and *last* must not be iterators into the vector, or the behavior is undefined. Any insertion operation can be expensive, see [vector Class](#) for a discussion of `vector` performance.

Example

```

// vector_insert.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 30 );

    cout << "v1 =" ;
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    v1.insert( v1.begin( ) + 1, 40 );
    cout << "v1 =";
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;
    v1.insert( v1.begin( ) + 2, 4, 50 );

    cout << "v1 =";
    for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
        cout << " " << *Iter;
    cout << endl;

    const auto v2 = v1;
    v1.insert( v1.begin( )+1, v2.begin( )+2, v2.begin( )+4 );
    cout << "v1 =";
    for (Iter = v1.begin( ); Iter != v1.end( ); Iter++ )
        cout << " " << *Iter;
    cout << endl;

    // initialize a vector of vectors by moving v1
    vector< vector<int> > vv1;

    vv1.insert( vv1.begin(), move( v1 ) );
    if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
    {
        cout << "vv1[0] =";
        for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
            cout << " " << *Iter;
        cout << endl;
    }
}

```

```

v1 = 10 20 30
v1 = 10 40 20 30
v1 = 10 40 50 50 50 50 20 30
v1 = 10 50 50 40 50 50 50 50 20 30
vv1[0] = 10 50 50 40 50 50 50 50 20 30

```

vector::iterator

A type that provides a random-access iterator that can read or modify any element in a vector.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for [begin](#).

vector::max_size

Returns the maximum length of the vector.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the vector.

Example

```
// vector_max_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector <int> v1;
    vector <int>::size_type i;

    i = v1.max_size( );
    cout << "The maximum possible length of the vector is " << i << "." << endl;
}
```

vector::operator[]

Returns a reference to the vector element at a specified position.

```
reference operator[](size_type Pos);

const_reference operator[](size_type Pos) const;
```

Parameters

PARAMETER	DESCRIPTION
<i>Pos</i>	The position of the vector element.

Return Value

If the position specified is greater than or equal to the size of the container, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `operator[]` is assigned to a reference, the vector object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error occurs if you attempt to access an element outside the bounds of the vector. See [Checked Iterators](#) for more information.

Example

```
// vector_op_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector <int> v1;

    v1.push_back( 10 );
    v1.push_back( 20 );

    int& i = v1[1];
    cout << "The second integer of v1 is " << i << endl;
}
```

vector::operator=

Replaces the elements of the vector with a copy of another vector.

```
vector& operator=(const vector& right);

vector& operator=(vector&& right);
```

Parameters

PARAMETER	DESCRIPTION
<i>right</i>	The vector being copied into the <code>vector</code> .

Remarks

After erasing any existing elements in a `vector`, `operator=` either copies or moves the contents of *right* into the `vector`.

Example

```

// vector_operator_as.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1, v2, v3;
    vector<int>::iterator iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

```

vector::pointer

A type that provides a pointer to an element in a vector.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

Example

```
// vector_pointer.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v;
    v.push_back( 11 );
    v.push_back( 22 );

    vector<int>::pointer ptr = &v[0];
    cout << *ptr << endl;
    ptr++;
    cout << *ptr << endl;
    *ptr = 44;
    cout << *ptr << endl;
}
```

```
11
22
44
```

vector::pop_back

Deletes the element at the end of the vector.

```
void pop_back();
```

Remarks

For a code example, see [vector::push_back\(\)](#).

vector::push_back

Adds an element to the end of the vector.

```
void push_back(const T& Val);

void push_back(T&& Val);
```

Parameters

Val

The value to assign to the element added to the end of the vector.

Example

```

// compile with: /EHsc /W4
#include <vector>
#include <iostream>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << " " << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(10 + i);
    }

    cout << "vector data: " << endl;
    print_collection(v);

    // pop_back() until it's empty, printing the last element as we go
    while (v.begin() != v.end()) {
        cout << "v.back(): "; print_elem(v.back()); cout << endl;
        v.pop_back();
    }
}

```

vector::rbegin

Returns an iterator to the first element in a reversed vector.

```

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;

```

Return Value

A reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed vector.

Remarks

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the vector object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the vector object can be modified.

Example

```
// vector_rbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator v1_Iter;
    vector<int>::reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    v1_Iter = v1.begin( );
    cout << "The first element of vector is "
         << *v1_Iter << "." << endl;

    v1_rIter = v1.rbegin( );
    cout << "The first element of the reversed vector is "
         << *v1_rIter << "." << endl;
}
```

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

vector::reference

A type that provides a reference to an element stored in a vector.

```
typedef typename Allocator::reference reference;
```

Example

See [at](#) for an example of how to use **reference** in the vector class.

vector::rend

Returns an iterator that addresses the location succeeding the last element in a reversed vector.

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed vector (the location that had preceded the first element in the unreversed vector).

Remarks

`rend` is used with a reversed vector just as [end](#) is used with a vector.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the vector object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the vector object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its vector.

The value returned by `rend` should not be dereferenced.

Example

```
// vector_rend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::reverse_iterator v1_rIter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
        cout << *v1_rIter << endl;
}
```

```
2
1
```

vector::reserve

Reserves a minimum length of storage for a vector object, allocating space if necessary.

```
void reserve(size_type count);
```

Parameters

count

The minimum length of storage to be allocated for the vector.

Example

```
// vector_reserve.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    //vector<int>::iterator Iter;

    v1.push_back( 1 );
    cout << "Current capacity of v1 = "
        << v1.capacity( ) << endl;
    v1.reserve( 20 );
    cout << "Current capacity of v1 = "
        << v1.capacity( ) << endl;
}
```

```
Current capacity of v1 = 1
Current capacity of v1 = 20
```

vector::resize

Specifies a new size for a vector.

```
void resize(size_type Newsize);
void resize(size_type Newsize, Type Val);
```

Parameters

Newsize

The new size of the vector.

Val

The initialization value of new elements added to the vector if the new size is larger than the original size. If the value is omitted, the new objects use their default constructor.

Remarks

If the container's size is less than the requested size, *Newsize*, elements are added to the vector until it reaches the requested size. If the container's size is larger than the requested size, the elements closest to the end of the container are deleted until the container reaches the size *Newsize*. If the present size of the container is the same as the requested size, no action is taken.

[size](#) reflects the current size of the vector.

Example

```
// vectorsizing.cpp
// compile with: /EHsc /W4
// Illustrates vector::reserve, vector::max_size,
// vector::resize, vector::resize, and vector::capacity.
//
// Functions:
//
//   vector::max_size - Returns maximum number of elements vector could
//                       hold.
//
//   vector::capacity - Returns number of elements for which memory has
//                       been allocated.
//
//   vector::size - Returns number of elements in the vector.
//
//   vector::resize - Reallocates memory for vector, preserves its
//                     contents if new size is larger than existing size.
//
//   vector::reserve - Allocates elements for vector to ensure a minimum
//                       size, preserving its contents if the new size is
//                       larger than existing size.
//
//   vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
// //////////////////////////////////////

// The debugger cannot handle symbols more than 255 characters long.
// The C++ Standard Library often creates symbols longer than that.
// The warning can be disabled:
// #pragma warning(disable:4786)
```

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }
    cout << endl;
}

void printvstats(const vector<int>& v) {
    cout << "    the vector's size is: " << v.size() << endl;
    cout << "    the vector's capacity is: " << v.capacity() << endl;
    cout << "    the vector's maximum size is: " << v.max_size() << endl;
}

int main()
{
    // declare a vector that begins with 0 elements.
    vector<int> v;

    // Show statistics about vector.
    cout << endl << "After declaring an empty vector:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    // Add one element to the end of the vector.
    v.push_back(-1);
    cout << endl << "After adding an element:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }
    cout << endl << "After adding 10 elements:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    v.resize(6);
    cout << endl << "After resizing to 6 elements without an initialization value:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    v.resize(9, 999);
    cout << endl << "After resizing to 9 elements with an initialization value of 999:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    v.resize(12);
    cout << endl << "After resizing to 12 elements without an initialization value:" << endl;
    printvstats(v);
    print("    the vector's contents: ", v);

    // Ensure there's room for at least 1000 elements.
    v.reserve(1000);
    cout << endl << "After vector::reserve(1000):" << endl;
    printvstats(v);

    // Ensure there's room for at least 2000 elements.
    v.resize(2000);
    cout << endl << "After vector::resize(2000):" << endl;
    printvstats(v);
}

```

vector::reverse_iterator

A type that provides a random-access iterator that can read or modify any element in a reversed vector.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the vector in reverse.

Example

See the example for [rbegin](#).

vector::shrink_to_fit

Discards excess capacity.

```
void shrink_to_fit();
```

Example

```
// vector_shrink_to_fit.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    //vector<int>::iterator Iter;

    v1.push_back( 1 );
    cout << "Current capacity of v1 = "
         << v1.capacity( ) << endl;
    v1.reserve( 20 );
    cout << "Current capacity of v1 = "
         << v1.capacity( ) << endl;
    v1.shrink_to_fit();
    cout << "Current capacity of v1 = "
         << v1.capacity( ) << endl;
}
```

```
Current capacity of v1 = 1
Current capacity of v1 = 20
Current capacity of v1 = 1
```

vector::size

Returns the number of elements in the vector.

```
size_type size() const;
```

Return Value

The current length of the vector.

Example

```
// vector_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1;
    vector<int>::size_type i;

    v1.push_back( 1 );
    i = v1.size( );
    cout << "Vector length is " << i << "." << endl;

    v1.push_back( 2 );
    i = v1.size( );
    cout << "Vector length is now " << i << "." << endl;
}
```

```
Vector length is 1.
Vector length is now 2.
```

vector::size_type

A type that counts the number of elements in a vector.

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for [capacity](#).

vector::swap

Exchanges the elements of two vectors.

```
void swap(
    vector<Type, Allocator>& right);

friend void swap(
    vector<Type, Allocator>& left,
    vector<Type, Allocator>& right);
```

Parameters

right

A vector providing the elements to be swapped, or a vector whose elements are to be exchanged with those of the vector *left*.

left

A vector whose elements are to be exchanged with those of the vector *right*.

Example

```

// vector_swap.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v1, v2;

    v1.push_back( 1 );
    v1.push_back( 2 );
    v1.push_back( 3 );

    v2.push_back( 10 );
    v2.push_back( 20 );

    cout << "The number of elements in v1 = " << v1.size( ) << endl;
    cout << "The number of elements in v2 = " << v2.size( ) << endl;
    cout << endl;

    v1.swap( v2 );

    cout << "The number of elements in v1 = " << v1.size( ) << endl;
    cout << "The number of elements in v2 = " << v2.size( ) << endl;
}

```

```

The number of elements in v1 = 3
The number of elements in v2 = 2

The number of elements in v1 = 2
The number of elements in v2 = 3

```

vector::value_type

A type that represents the data type stored in a vector.

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Type`.

Example

```

// vector_value_type.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int>::value_type AnInt;
    AnInt = 44;
    cout << AnInt << endl;
}

```

vector::vector

Constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other vector.

```
vector();
explicit vector(const Allocator& Al);
explicit vector(size_type Count);
vector(size_type Count, const Type& Val);
vector(size_type Count, const Type& Val, const Allocator& Al);

vector(const vector& Right);
vector(vector&& Right);
vector(initializer_list<Type> IList, const _Allocator& Al);

template <class InputIterator>
vector(InputIterator First, InputIterator Last);
template <class InputIterator>
vector(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

PARAMETER	DESCRIPTION
<i>Al</i>	The allocator class to use with this object. get_allocator returns the allocator class for the object.
<i>Count</i>	The number of elements in the constructed vector.
<i>Val</i>	The value of the elements in the constructed vector.
<i>Right</i>	The vector of which the constructed vector is to be a copy.
<i>First</i>	Position of the first element in the range of elements to be copied.
<i>Last</i>	Position of the first element beyond the range of elements to be copied.
<i>IList</i>	The initializer_list containing the elmeents to copy.

Remarks

All constructors store an allocator object (*Al*) and initialize the vector.

The first two constructors specify an empty initial vector. The second explicitly specifies the allocator type (*Al*) to be used.

The third constructor specifies a repetition of a specified number (*Count*) of elements of the default value for class `Type`.

The fourth and fifth constructors specify a repetition of (*Count*) elements of value *Val*.

The sixth constructor specifies a copy of the vector *Right*.

The seventh constructor moves the vector *Right*.

The eighth constructor uses an initializer_list to specify the elements.

The ninth and tenth constructors copy the range [`First` , `Last`) of a vector.

Example

```
// vector_ctor.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int>::iterator v1_Iter, v2_Iter, v3_Iter, v4_Iter, v5_Iter, v6_Iter;

    // Create an empty vector v0
    vector<int> v0;

    // Create a vector v1 with 3 elements of default value 0
    vector<int> v1(3);

    // Create a vector v2 with 5 elements of value 2
    vector<int> v2(5, 2);

    // Create a vector v3 with 3 elements of value 1 and with the allocator
    // of vector v2
    vector<int> v3(3, 1, v2.get_allocator());

    // Create a copy, vector v4, of vector v2
    vector<int> v4(v2);

    // Create a new temporary vector for demonstrating copying ranges
    vector<int> v5(5);
    for (auto i : v5) {
        v5[i] = i;
    }

    // Create a vector v6 by copying the range v5[ first, last)
    vector<int> v6(v5.begin() + 1, v5.begin() + 3);

    cout << "v1 =";
    for (auto& v : v1){
        cout << " " << v;
    }
    cout << endl;

    cout << "v2 =";
    for (auto& v : v2){
        cout << " " << v;
    }
    cout << endl;

    cout << "v3 =";
    for (auto& v : v3){
        cout << " " << v;
    }
    cout << endl;
    cout << "v4 =";
    for (auto& v : v4){
        cout << " " << v;
    }
    cout << endl;

    cout << "v5 =";
    for (auto& v : v5){
        cout << " " << v;
    }
    cout << endl;

    cout << "v6 =";
    for (auto& v : v6){
```



```

        cout << " " << v;
    }
    cout << endl;

    // Move vector v2 to vector v7
    vector<int> v7(move(v2));
    vector<int>::iterator v7_Iter;

    cout << "v7 =";
    for (auto& v : v7){
        cout << " " << v;
    }
    cout << endl;

    vector<int> v8{ { 1, 2, 3, 4 } };
    for (auto& v : v8){
        cout << " " << v ;
    }
    cout << endl;
}

```

```

v1 = 0 0 0v2 = 2 2 2 2 2v3 = 1 1 1v4 = 2 2 2 2 2v5 = 0 1 2 3 4v6 = 1 2v7 = 2 2 2 2 21 2 3 4

```

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

vector<bool> Class

11/9/2018 • 7 minutes to read • [Edit Online](#)

The `vector<bool>` class is a partial specialization of `vector` for elements of type **bool**. It has an allocator for the underlying type that's used by the specialization, which provides space optimization by storing one **bool** value per bit.

Syntax

```
template <class Allocator = allocator<bool>>
class vector<bool, Allocator>
```

Remarks

This class template specialization behaves like `vector`, except for the differences explained in this article.

Operations that deal with the **bool** type correspond to values in the container storage.

`allocator_traits::construct` is not used to construct these values.

Typedefs

TYPE NAME	DESCRIPTION
<code>const_pointer</code>	A typedef to a <code>const_iterator</code> that can serve as a constant pointer to a Boolean element of the <code>vector<bool></code> .
<code>const_reference</code>	A typedef for bool . After initialization, it does not observe updates to the original value.
<code>pointer</code>	A typedef to an <code>iterator</code> that can serve as a pointer to a Boolean element of the <code>vector<bool></code> .

Member functions

MEMBER FUNCTION	DESCRIPTION
<code>flip</code>	Reverses all bits in the <code>vector<bool></code> .
<code>swap</code>	Exchanges the elements of two <code>vector<bool></code> s.
<code>operator[]</code>	Returns a simulated reference to the <code>vector<bool></code> element at a specified position.
<code>at</code>	Functions the same as the unspecialized <code>vector::at</code> function, except that it uses the proxy class <code>vector<bool>::reference</code> . Also see <code>operator[]</code> .
<code>front</code>	Functions the same as the unspecialized <code>vector::front</code> function, except that it uses the proxy class <code>vector<bool>::reference</code> . Also see <code>operator[]</code> .

MEMBER FUNCTION	DESCRIPTION
<code>back</code>	Functions the same as the unspecialized <code>vector::back</code> function, except that it uses the proxy class <code>vector<bool>::reference</code> . Also see <code>operator[]</code> .

Proxy Class

<code>vector<bool>::reference</code> Class	A class that acts as a proxy to simulate <code>bool&</code> behavior, and whose objects can provide references to elements (single bits) within a <code>vector<bool></code> object.
--	---

Requirements

Header: `<vector>`

Namespace: `std`

`vector<bool>::const_pointer`

A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the `vector<bool>` object.

```
typedef const_iterator const_pointer;
```

`vector<bool>::const_reference`

A type that describes an object that can serve as a constant reference to a Boolean element of the sequence contained by the `vector<bool>` object.

```
typedef bool const_reference;
```

Remarks

For more information and code examples, see `vector<bool>::reference::operator=`.

`vector<bool>::flip`

Reverses all bits in a `vector<bool>`.

```
void flip();
```

Example

```

// vector_bool_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha; // format output for subsequent code

    vector<bool> vb = { true, false, false, true, true };
    cout << "The vector is:" << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vb.flip();

    cout << "The flipped vector is:" << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;
}

```

vector<bool>::operator[]

Returns a simulated reference to the `vector<bool>` element at a specified position.

```

vector<bool>::reference operator[](size_type Pos);

vector<&bool&>::const_reference operator[](size_type Pos) const;

```

Parameters

PARAMETER	DESCRIPTION
<i>Pos</i>	The position of the <code>vector<bool></code> element.

Return Value

A `vector<bool>::reference` or `vector<bool>::const_reference` object that contains the value of the indexed element.

If the position specified is greater than or equal to the size of the container, the result is undefined.

Remarks

If you compile with `_ITERATOR_DEBUG_LEVEL` set, a run-time error occurs if you attempt to access an element outside the bounds of the vector. For more information, see [Checked Iterators](#).

Example

This code example shows the correct use of `vector<bool>::operator[]` and two common coding mistakes, which are commented out. These mistakes cause errors because the address of the `vector<bool>::reference` object that `vector<bool>::operator[]` returns cannot be taken.

```

// cl.exe /EHsc /nologo /W4 /MTd
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;
    vector<bool> vb;

    vb.push_back(true);
    vb.push_back(false);

    //    bool* pb = &vb[1]; // conversion error - do not use
    //    bool& refb = vb[1]; // conversion error - do not use
    bool hold = vb[1];
    cout << "The second element of vb is " << vb[1] << endl;
    cout << "The held value from the second element of vb is " << hold << endl;

    // Note this doesn't modify hold.
    vb[1] = true;
    cout << "The second element of vb is " << vb[1] << endl;
    cout << "The held value from the second element of vb is " << hold << endl;
}

```

vector<bool>::pointer

A type that describes an object that can serve as a pointer to a Boolean element of the sequence contained by the `vector<bool>` object.

```
typedef iterator pointer;
```

vector<bool>::reference Class

The `vector<bool>::reference` class is a proxy class provided by the [vector<bool> Class](#) to simulate `bool&`.

Remarks

A simulated reference is required because C++ does not natively allow direct references to bits. `vector<bool>` uses only one bit per element, which can be referenced by using this proxy class. However, the reference simulation is not complete because certain assignments are not valid. For example, because the address of the `vector<bool>::reference` object cannot be taken, the following code that uses [vector<bool>::operator\[\]](#) is not correct:

```

vector<bool> vb;
//...
bool* pb = &vb[1]; // conversion error - do not use
bool& refb = vb[1]; // conversion error - do not use

```

vector<bool>::reference::flip

Inverts the Boolean value of a referenced [vector<bool>](#) element.

```
void flip();
```

Example

```
// vector_bool_ref_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    cout << "The vector is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vector<bool>::reference vbref = vb.front();
    vbref.flip();

    cout << "The vector with first element flipped is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;
}
```

```
The vector is:
true false false true true
The vector with first element flipped is:
false false false true true
```

vector<bool>::reference::operator bool

Provides an implicit conversion from `vector<bool>::reference` to **bool**.

```
operator bool() const;
```

Return Value

The Boolean value of the element of the vector<bool> object.

Remarks

The `vector<bool>` object cannot be modified by this operator.

vector<bool>::reference::operator=

Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

```
reference& operator=(const reference& Right);
reference& operator=(bool Val);
```

Parameters

Right

The element reference whose value is to be assigned to the bit.

Val

The Boolean value to be assigned to the bit.

Example

```

// vector_bool_ref_op_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    print("The vector is: ", vb);

    // Invoke vector<bool>::reference::operator=()
    vector<bool>::reference refelem1 = vb[0];
    vector<bool>::reference refelem2 = vb[1];
    vector<bool>::reference refelem3 = vb[2];

    bool b1 = refelem1;
    bool b2 = refelem2;
    bool b3 = refelem3;
    cout << "The original value of the 1st element stored in a bool: " << b1 << endl;
    cout << "The original value of the 2nd element stored in a bool: " << b2 << endl;
    cout << "The original value of the 3rd element stored in a bool: " << b3 << endl;
    cout << endl;

    refelem2 = refelem1;

    print("The vector after assigning refelem1 to refelem2 is now: ", vb);

    refelem3 = true;

    print("The vector after assigning false to refelem1 is now: ", vb);

    // The initial values are still stored in the bool variables and remained unchanged
    cout << "The original value of the 1st element still stored in a bool: " << b1 << endl;
    cout << "The original value of the 2nd element still stored in a bool: " << b2 << endl;
    cout << "The original value of the 3rd element still stored in a bool: " << b3 << endl;
    cout << endl;
}

```

```

The vector is: true false false true true
The original value of the 1st element stored in a bool: true
The original value of the 2nd element stored in a bool: false
The original value of the 3rd element stored in a bool: false

The vector after assigning refelem1 to refelem2 is now: true true false true true
The vector after assigning false to refelem1 is now: true true true true true
The original value of the 1st element still stored in a bool: true
The original value of the 2nd element still stored in a bool: false
The original value of the 3rd element still stored in a bool: false

```

vector<bool>::swap

Static member function that exchanges two elements of Boolean vectors (`vector<bool>`) by using the proxy class [vector<bool>::reference](#).

```
static void swap(  
    reference Left,  
    reference Right);
```

Parameters

Left

The element to be exchanged with the *Right* element.

Right

The element to be exchanged with the *Left* element.

Remarks

This overload supports the special proxy requirements of `vector<bool>`. [vector::swap](#) has the same functionality as the single-argument overload of `vector<bool>::swap()`.

See also

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

vector<bool>::reference Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `vector<bool>::reference` class is a proxy class provided by the [vector<bool> Class](#) to simulate `bool&`.

Remarks

A simulated reference is required because C++ does not natively allow direct references to bits. `vector<bool>` uses only one bit per element, which can be referenced by using this proxy class. However, the reference simulation is not complete because certain assignments are not valid. For example, because the address of the `vector<bool>::reference` object cannot be taken, the following code that tries to use `vector<bool>::operator&` is not correct:

```
vector<bool> vb;
// ...
bool* pb = &vb[1]; // conversion error - do not use
bool& refb = vb[1]; // conversion error - do not use
```

Member functions

MEMBER FUNCTION	DESCRIPTION
flip	Inverts the Boolean value of a vector element.
operator bool	Provides an implicit conversion from <code>vector<bool>::reference</code> to bool .
operator=	Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

Requirements

Header: `<vector>`

Namespace: `std`

See also

[<vector>](#)

[Thread Safety in the C++ Standard Library](#)

[C++ Standard Library Reference](#)

vector<bool>::reference::flip

11/9/2018 • 2 minutes to read • [Edit Online](#)

Inverts the Boolean value of a referenced `vector<bool>` element.

Syntax

```
void flip();
```

Example

```
// vector_bool_ref_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    cout << "The vector is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vector<bool>::reference vbref = vb.front();
    vbref.flip();

    cout << "The vector with first element flipped is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;
}
```

Output

```
The vector is:
true false false true true
The vector with first element flipped is:
false false false true true
```

Requirements

Header: <vector>

Namespace: std

See also

[vector<bool>::reference Class](#)

[C++ Standard Library Reference](#)

vector<bool>::reference::operator bool

10/31/2018 • 2 minutes to read • [Edit Online](#)

Provides an implicit conversion from `vector<bool>::reference` to **bool**.

Syntax

```
operator bool() const;
```

Return Value

The Boolean value of the element of the `vector<bool>` object.

Remarks

The `vector<bool>` object cannot be modified by this operator.

Requirements

Header: <vector>

Namespace: std

See also

[vector<bool>::reference Class](#)
[C++ Standard Library Reference](#)

vector<bool>::reference::operator=

11/9/2018 • 2 minutes to read • [Edit Online](#)

Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

Syntax

```
reference& operator=(const reference& Right);  
  
reference& operator=(bool Val);
```

Parameters

Right

The element reference whose value is to be assigned to the bit.

Val

The Boolean value to be assigned to the bit.

Example

```

// vector_bool_ref_op_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    print("The vector is: ", vb);

    // Invoke vector<bool>::reference::operator=()
    vector<bool>::reference refelem1 = vb[0];
    vector<bool>::reference refelem2 = vb[1];
    vector<bool>::reference refelem3 = vb[2];

    bool b1 = refelem1;
    bool b2 = refelem2;
    bool b3 = refelem3;
    cout << "The original value of the 1st element stored in a bool: " << b1 << endl;
    cout << "The original value of the 2nd element stored in a bool: " << b2 << endl;
    cout << "The original value of the 3rd element stored in a bool: " << b3 << endl;
    cout << endl;

    refelem2 = refelem1;

    print("The vector after assigning refelem1 to refelem2 is now: ", vb);

    refelem3 = true;

    print("The vector after assigning false to refelem1 is now: ", vb);

    // The initial values are still stored in the bool variables and remained unchanged
    cout << "The original value of the 1st element still stored in a bool: " << b1 << endl;
    cout << "The original value of the 2nd element still stored in a bool: " << b2 << endl;
    cout << "The original value of the 3rd element still stored in a bool: " << b3 << endl;
    cout << endl;
}

```

Output

```
The vector is: true false false true true
The original value of the 1st element stored in a bool: true
The original value of the 2nd element stored in a bool: false
The original value of the 3rd element stored in a bool: false

The vector after assigning refelem1 to refelem2 is now: true true false true true
The vector after assigning false to refelem1 is now: true true true true true
The original value of the 1st element still stored in a bool: true
The original value of the 2nd element still stored in a bool: false
The original value of the 3rd element still stored in a bool: false
```

Requirements

Header: <vector>

Namespace: std

See also

[vector<bool>::reference Class](#)

[C++ Standard Library Reference](#)

<vector> functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

swap

Exchanges the elements of two vectors.

```
template <class Type, class Allocator>
void swap(vector<Type, Allocator>& left, vector<Type, Allocator>& right);
```

Parameters

right

The vector providing the elements to be swapped, or the vector whose elements are to be exchanged with those of the vector *left*.

left

The vector whose elements are to be exchanged with those of the vector *right*.

Remarks

The template function is an algorithm specialized on the container class `vector` to execute the member function `left.vector::swap (right)`. These are instances of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template < class T> void swap(T&, T&)**, in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function `vector::swap` for an example that uses the template version of `swap`.

See also

[<vector>](#)

<vector> operators

10/31/2018 • 4 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator></code>	<code>operator>=</code>
<code>operator<</code>	<code>operator<=</code>	<code>operator==</code>

operator!=

Tests if the object on the left side of the operator is not equal to the object on the right side.

```
bool operator!=(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vectors are not equal; **false** if the vectors are equal.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// vector_op_ne.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector<int> v1, v2;
    v1.push_back( 1 );
    v2.push_back( 2 );

    if ( v1 != v2 )
        cout << "Vectors not equal." << endl;
    else
        cout << "Vectors equal." << endl;
}
```

Vectors not equal.

operator<

Tests if the object on the left side of the operator is less than the object on the right side.

```
bool operator<(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vector on the left side of the operator is less than the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_lt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector <int> v1, v2;
    v1.push_back( 1 );
    v1.push_back( 2 );
    v1.push_back( 4 );

    v2.push_back( 1 );
    v2.push_back( 3 );

    if ( v1 < v2 )
        cout << "Vector v1 is less than vector v2." << endl;
    else
        cout << "Vector v1 is not less than vector v2." << endl;
}
```

```
Vector v1 is less than vector v2.
```

operator<=

Tests if the object on the left side of the operator is less than or equal to the object on the right side.

```
bool operator<=(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vector on the left side of the operator is less than or equal to the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_le.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector <int> v1, v2;
    v1.push_back( 1 );
    v1.push_back( 2 );
    v1.push_back( 4 );

    v2.push_back( 1 );
    v2.push_back( 3 );

    if ( v1 <= v2 )
        cout << "Vector v1 is less than or equal to vector v2." << endl;
    else
        cout << "Vector v1 is greater than vector v2." << endl;
}
```

Vector v1 is less than or equal to vector v2.

operator==

Tests if the object on the left side of the operator is equal to the object on the right side.

```
bool operator==(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vector on the left side of the operator is equal to the vector on the right side of the operator; otherwise **false**.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
// vector_op_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector<int> v1, v2;
    v1.push_back( 1 );
    v2.push_back( 1 );

    if ( v1 == v2 )
        cout << "Vectors equal." << endl;
    else
        cout << "Vectors not equal." << endl;
}
```

Vectors equal.

operator>

Tests if the object on the left side of the operator is greater than the object on the right side.

```
bool operator>(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vector on the left side of the operator is greater than the vector on the right side of the operator; otherwise **false**.

Example

```
// vector_op_gt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector <int> v1, v2;
    v1.push_back( 1 );
    v1.push_back( 3 );
    v1.push_back( 1 );

    v2.push_back( 1 );
    v2.push_back( 2 );
    v2.push_back( 2 );

    if ( v1 > v2 )
        cout << "Vector v1 is greater than vector v2." << endl;
    else
        cout << "Vector v1 is not greater than vector v2." << endl;
}
```

Vector v1 is greater than vector v2.

operator>=

Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

```
bool operator>=(const vector<Type, Allocator>& left, const vector<Type, Allocator>& right);
```

Parameters

left

An object of type `vector`.

right

An object of type `vector`.

Return Value

true if the vector on the left side of the operator is greater than or equal to the vector on the right side of the vector; otherwise **false**.

Example

```
// vector_op_ge.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;

    vector<int> v1, v2;
    v1.push_back( 1 );
    v1.push_back( 3 );
    v1.push_back( 1 );

    v2.push_back( 1 );
    v2.push_back( 2 );
    v2.push_back( 2 );

    if ( v1 >= v2 )
        cout << "Vector v1 is greater than or equal to vector v2." << endl;
    else
        cout << "Vector v1 is less than vector v2." << endl;
}
```

Vector v1 is greater than or equal to vector v2.

See also

[<vector>](#)

C++ Standard Library Overview

10/31/2018 • 2 minutes to read • [Edit Online](#)

All C++ library entities are declared or defined in one or more standard headers. This implementation includes two additional headers, `<hash_map>` and `<hash_set>`, that are not required by the C++ Standard. For a complete list of headers that this implementation supports, see [Header Files Reference](#).

A freestanding implementation of the C++ library provides only a subset of these headers:

<code><cstddef></code>	<code><cstdlib></code> (declaring at least the functions <code>abort</code> , <code>atexit</code> , and <code>exit</code>)
<code><exception></code>	<code><limits></code>
<code><new></code>	<code><cstdarg></code>

The C++ library headers have two broader subdivisions:

- [iostreams](#) conventions.
- [C++ Standard Library Reference](#) conventions.

This section contains the following sections:

- [Using C++ Library Headers](#)
- [C++ Library Conventions](#)
- [iostreams Conventions](#)
- [C++ Program Startup and Termination](#)
- [Safe Libraries: C++ Standard Library](#)
- [Checked Iterators](#)
- [Debug Iterator Support](#)
- [C++ Standard Library Reference](#)
- [Thread Safety in the C++ Standard Library](#)
- [stdext Namespace](#)
- [Regular Expressions \(C++\)](#)

For more information about Visual C++ run-time libraries, see [CRT Library Features](#).

See also

[C++ Standard Library](#)

Using C++ Library Headers

10/31/2018 • 2 minutes to read • [Edit Online](#)

You include the contents of a standard header by naming it in an include directive.

```
#include <iostream> // include I/O facilities
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro with the same name as the function that masks the function declaration and achieves the same effect. For more information on masking macros, see [C++ Library Conventions](#).

All names other than **operator delete** and **operator new** in the C++ library headers are defined in the `std` namespace, or in a namespace nested within the `std` namespace. You refer to the name `cin`, for example, as `std::cin`. Note, however, that macro names are not subject to namespace qualification, so you always write `__STD_COMPLEX` without a namespace qualifier.

In some translation environments, including a C++ library header may hoist external names declared in the `std` namespace into the global namespace as well, with individual **using** declarations for each of the names. Otherwise, the header does *not* introduce any library names into the current namespace.

The C++ Standard requires that the C Standard headers declare all external names in namespace `std`, then hoist them into the global namespace with individual **using** declarations for each of the names. But in some translation environments the C Standard headers include no namespace declarations, declaring all names directly in the global namespace. Thus, the most portable way to deal with namespaces is to follow two rules:

- To assuredly declare in namespace `std` an external name that is traditionally declared in `<stdlib.h>`, for example, include the header `<cstdlib>`. Know that the name might also be declared in the global namespace.
- To assuredly declare in the global namespace an external name declared in `<stdlib.h>`, include the header `<stdlib.h>` directly. Know that the name might also be declared in namespace `std`.

Thus, if you want to call `std::abort` to cause abnormal termination, you should include `<cstdlib>`. If you want to call `abort`, you should include `<stdlib.h>`.

Alternatively, you can write the declaration:

```
using namespace std;
```

which brings all library names into the current namespace. If you write this declaration immediately after all include directives, you hoist the names into the global namespace. You can subsequently ignore namespace considerations in the remainder of the translation unit. You also avoid most differences across different translation

environments.

Unless specifically indicated otherwise, you may not define names in the `std` namespace, or in a namespace nested within the `std` namespace, within your program.

See also

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

C++ Library Conventions

10/31/2018 • 2 minutes to read • [Edit Online](#)

The C++ library obeys much the same conventions as the Standard C Library, plus a few more outlined here.

An implementation has certain latitude in how it declares types and functions in the C++ library:

- Names of functions in the Standard C library may have either extern "C++" or extern "C" linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (nonvirtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different integer types.
- A bitmask type can be implemented as either an integer type or an enumeration. In either case, you can perform bitwise operations (such as `AND` and `OR`) on values of the same bitmask type. The elements `A` and `B` of a bitmask type are nonzero values such that `A & B` is zero.
- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions:

- The Standard C Library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.
- A library function name outside a class will not have additional, undocumented, function signatures. You can reliably take its address.
- Base classes and member functions described as virtual are assuredly virtual, while those described as nonvirtual are assuredly nonvirtual.
- Two types defined by the C++ library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions, can throw *at most* those exceptions listed in any exception specification. No destructors supplied by the library throw exceptions. Functions in the Standard C Library may propagate an exception, as when `qsort` calls a comparison function that throws an exception, but they do not otherwise throw exceptions.

See also

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

C++ Program Startup and Termination

10/31/2018 • 2 minutes to read • [Edit Online](#)

A C++ program performs the same operations as a C program does at program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function `main`, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that some [iostreams](#) objects are properly initialized for use by these static constructors. These control text streams are:

- `cin` — for standard input.
- `cout` — for standard output.
- `cerr` — for unbuffered standard error output.
- `clog` — for buffered standard error output.

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from `main` or calling `exit` calls all functions registered with `atexit` in reverse order of registry. An exception thrown from such a registered function calls `terminate`.

See also

[C++ Standard Library Overview](#)

[Thread Safety in the C++ Standard Library](#)

Safe Libraries: C++ Standard Library

5/7/2019 • 2 minutes to read • [Edit Online](#)

Several enhancements have been made to the libraries that ship with Microsoft C++, including the C++ Standard Library, to make them more secure.

Several methods in the C++ Standard Library have been identified as potentially unsafe because they could lead to a buffer overrun or other code defect. The use of these methods is discouraged, and new, more secure methods have been created to replace them. These new methods all end in `_s`.

Several enhancements have also been made to make iterators and algorithms more secure. For more information, see [Checked Iterators](#), [Debug Iterator Support](#) and `_ITERATOR_DEBUG_LEVEL`.

Remarks

The following table lists the C++ Standard Library methods that are potentially unsafe, as well as their safer equivalent:

POTENTIALLY UNSAFE METHOD	SAFER EQUIVALENT
<code>copy</code>	<code>basic_string::_Copy_s</code>
<code>copy</code>	<code>char_traits::_Copy_s</code>

If you call any one of the potentially unsafe methods above, or if you use iterators incorrectly, the compiler will generate [Compiler Warning \(level 3\) C4996](#). For information on how to disable these warnings, see `_SCL_SECURE_NO_WARNINGS`.

In This Section

[_ITERATOR_DEBUG_LEVEL](#)

[_SCL_SECURE_NO_WARNINGS](#)

[Checked Iterators](#)

[Debug Iterator Support](#)

See also

[C++ Standard Library Overview](#)

_ITERATOR_DEBUG_LEVEL

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `_ITERATOR_DEBUG_LEVEL` macro controls whether [checked iterators](#) and [debug iterator support](#) are enabled. This macro supersedes and combines the functionality of the older `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` macros.

Macro Values

The following table summarizes the possible values for the `_ITERATOR_DEBUG_LEVEL` macro.

COMPILATION MODE	MACRO VALUE	DESCRIPTION
Debug		
	0	Disables checked iterators and disables iterator debugging.
	1	Enables checked iterators and disables iterator debugging.
	2 (Default)	Enables iterator debugging; checked iterators are not relevant.
Release		
	0 (Default)	Disables checked iterators.
	1	Enables checked iterators; iterator debugging is not relevant.

In release mode, the compiler generates an error if you specify `_ITERATOR_DEBUG_LEVEL` as 2.

Remarks

The `_ITERATOR_DEBUG_LEVEL` macro controls whether [checked iterators](#) are enabled, and in Debug mode, whether [debug iterator support](#) is enabled. If `_ITERATOR_DEBUG_LEVEL` is defined as 1 or 2, checked iterators ensure that the bounds of your containers are not overwritten. If `_ITERATOR_DEBUG_LEVEL` is 0, iterators are not checked. When `_ITERATOR_DEBUG_LEVEL` is defined as 1, any unsafe iterator use causes a runtime error and the program is terminated. When `_ITERATOR_DEBUG_LEVEL` is defined as 2, unsafe iterator use causes an assert and a runtime error dialog that lets you break into the debugger.

Because the `_ITERATOR_DEBUG_LEVEL` macro supports similar functionality to the `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` macros, you may be uncertain which macro and macro value to use in a particular situation. To prevent confusion, we recommend that you use only the `_ITERATOR_DEBUG_LEVEL` macro. This table describes the equivalent `_ITERATOR_DEBUG_LEVEL` macro value to use for various values of `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` in existing code.

<code>_ITERATOR_DEBUG_LEVEL</code>	<code>_SECURE_SCL</code>	<code>_HAS_ITERATOR_DEBUGGING</code>
0 (Release default)	0 (disabled)	0 (disabled)
1	1 (enabled)	0 (disabled)
2 (Debug default)	(not relevant)	1 (enabled in Debug mode)

For information on how to disable warnings about checked iterators, see [_SCL_SECURE_NO_WARNINGS](#).

Example

To specify a value for the `_ITERATOR_DEBUG_LEVEL` macro, use a [/D](#) compiler option to define it on the command line, or use `#define` before the C++ Standard Library headers are included in your source files. For example, on the command line, to compile *sample.cpp* in debug mode and to use debug iterator support, you can specify the `_ITERATOR_DEBUG_LEVEL` macro definition:

```
c1 /EHsc /Zi /MDd /D_ITERATOR_DEBUG_LEVEL=1 sample.cpp
```

In a source file, specify the macro before any standard library headers that define iterators.

```
// sample.cpp

#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>

// ...
```

See also

[Checked Iterators](#)

[Debug Iterator Support](#)

[Safe Libraries: C++ Standard Library](#)

_SCL_SECURE_NO_WARNINGS

10/31/2018 • 2 minutes to read • [Edit Online](#)

Calling any of the potentially unsafe methods in the C++ Standard Library results in [Compiler Warning \(level 3\) C4996](#). To disable this warning, define the macro `_SCL_SECURE_NO_WARNINGS` in your code:

```
#define _SCL_SECURE_NO_WARNINGS
```

If you use precompiled headers, put this directive in your precompiled header file before you include any C runtime library or standard library headers. If you put it in an individual source code file before you include the precompiled header file, it is ignored by the compiler.

Remarks

Other ways to disable warning C4996 include:

- Using the `/D` ([Preprocessor Definitions](#)) compiler option:

```
cl /D_SCL_SECURE_NO_WARNINGS [other compiler options] myfile.cpp
```

- Using the `/w` compiler option:

```
cl /wd4996 [other compiler options] myfile.cpp
```

- Using the `#pragma warning` directive:

```
#pragma warning(disable:4996)
```

Also, you can manually change the level of warning C4996 with the `/w<l><n>` compiler option. For example, to set warning C4996 to level 4:

```
cl /w44996 [other compiler options] myfile.cpp
```

For more information, see [/w](#), [/W0](#), [/W1](#), [/W2](#), [/W3](#), [/W4](#), [/w1](#), [/w2](#), [/w3](#), [/w4](#), [/Wall](#), [/wd](#), [/we](#), [/wo](#), [/Wv](#), [/WX](#) ([Warning Level](#)).

See also

[Safe Libraries: C++ Standard Library](#)

Checked Iterators

11/9/2018 • 5 minutes to read • [Edit Online](#)

Checked iterators ensure that the bounds of your container are not overwritten. Checked iterators apply to both release builds and debug builds. For more information about how to use debug iterators when you compile in debug mode, see [Debug Iterator Support](#).

Remarks

For information about how to disable warnings that are generated by checked iterators, see [_SCL_SECURE_NO_WARNINGS](#).

You can use the [_ITERATOR_DEBUG_LEVEL](#) preprocessor macro to enable or disable the checked iterators feature. If [_ITERATOR_DEBUG_LEVEL](#) is defined as 1 or 2, unsafe use of iterators causes a runtime error and the program is terminated. If defined as 0, checked iterators are disabled. By default, the value for [_ITERATOR_DEBUG_LEVEL](#) is 0 for release builds and 2 for debug builds.

IMPORTANT

Older documentation and source code may refer to the [_SECURE_SCL](#) macro. Use [_ITERATOR_DEBUG_LEVEL](#) to control [_SECURE_SCL](#). For more information, see [_ITERATOR_DEBUG_LEVEL](#).

When [_ITERATOR_DEBUG_LEVEL](#) is defined as 1 or 2, these iterator checks are performed:

- All standard iterators (for example, [vector::iterator](#)) are checked.
- If an output iterator is a checked iterator, calls to standard library functions such as [std::copy](#) get checked behavior.
- If an output iterator is an unchecked iterator, calls to standard library functions cause compiler warnings.
- The following functions generate a runtime error if there is an access that is outside the bounds of the container:

basic_string::operator[]	bitset::operator[]	back	front
deque::operator[]	back	front	back
front	vector::operator[]	back	front

When [_ITERATOR_DEBUG_LEVEL](#) is defined as 0:

- All standard iterators are unchecked. Iterators can move beyond the container boundaries, which leads to undefined behavior.
- If an output iterator is a checked iterator, calls to standard library functions such as `std::copy` get checked behavior.
- If an output iterator is an unchecked iterator, calls to standard library functions get unchecked behavior.

A checked iterator refers to an iterator that calls `invalid_parameter_handler` if you attempt to move past the boundaries of the container. For more information about `invalid_parameter_handler`, see [Parameter Validation](#).

The iterator adaptors that support checked iterators are [checked_array_iterator Class](#) and [unchecked_array_iterator Class](#).

Example

When you compile by using `_ITERATOR_DEBUG_LEVEL` set to 1 or 2, a runtime error will occur if you attempt to access an element that is outside the bounds of the container by using the indexing operator of certain classes.

```
// checked_iterators_1.cpp
// cl.exe /Zi /MDd /EHsc /W4

#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v;
    v.push_back(67);

    int i = v[0];
    cout << i << endl;

    i = v[1]; //triggers invalid parameter handler
}
```

This program prints "67" then pops up an assertion failure dialog box with additional information about the failure.

Example

Similarly, when you compile by using `_ITERATOR_DEBUG_LEVEL` set to 1 or 2, a runtime error will occur if you attempt to access an element by using `front` or `back` in container classes when the container is empty.

```
// checked_iterators_2.cpp
// cl.exe /Zi /MDd /EHsc /W4
#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v;

    int& i = v.front(); // triggers invalid parameter handler
}
```

This program pops up an assertion failure dialog box with additional information about the failure.

Example

The following code demonstrates various iterator use-case scenarios with comments about each. By default, `_ITERATOR_DEBUG_LEVEL` is set to 2 in Debug builds, and to 0 in Retail builds.

```
// checked_iterators_3.cpp
// cl.exe /MTd /EHsc /W4

#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <numeric>
#include <string>
#include <vector>

using namespace std;

template <typename C>
void print(const string& s, const C& c)
{
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    vector<int> v(16);
    iota(v.begin(), v.end(), 0);
    print("v: ", v);

    // OK: vector::iterator is checked in debug mode
    // (i.e. an overrun causes a debug assertion)
    vector<int> v2(16);
    transform(v.begin(), v.end(), v2.begin(), [](int n) { return n * 2; });
    print("v2: ", v2);

    // OK: back_insert_iterator is marked as checked in debug mode
    // (i.e. an overrun is impossible)
    vector<int> v3;
    transform(v.begin(), v.end(), back_inserter(v3), [](int n) { return n * 3; });
    print("v3: ", v3);

    // OK: array::iterator is checked in debug mode
    // (i.e. an overrun causes a debug assertion)
    array<int, 16> a4;
    transform(v.begin(), v.end(), a4.begin(), [](int n) { return n * 4; });
    print("a4: ", a4);

    // OK: Raw arrays are checked in debug mode
    // (an overrun causes a debug assertion)
    // NOTE: This applies only when raw arrays are given to C++ Standard Library algorithms!
    int a5[16];
    transform(v.begin(), v.end(), a5, [](int n) { return n * 5; });
    print("a5: ", a5);

    // WARNING C4996: Pointers cannot be checked in debug mode
    // (an overrun causes undefined behavior)
    int a6[16];
    int * p6 = a6;
    transform(v.begin(), v.end(), p6, [](int n) { return n * 6; });
    print("a6: ", a6);
}
```

```

// OK: stdext::checked_array_iterator is checked in debug mode
// (an overrun causes a debug assertion)
int a7[16];
int * p7 = a7;
transform(v.begin(), v.end(), stdext::make_checked_array_iterator(p7, 16), [](int n) { return
n * 7; });
print("a7: ", a7);

// WARNING SILENCED: stdext::unchecked_array_iterator is marked as checked in debug mode
// (it performs no checking, so an overrun causes undefined behavior)
int a8[16];
int * p8 = a8;
transform(v.begin(), v.end(), stdext::make_unchecked_array_iterator(p8), [](int n) { return n
* 8; });
print("a8: ", a8);
}

```

When you compile this code by using `cl.exe /EHsc /W4 /MTd checked_iterators_3.cpp` the compiler emits a warning, but compiles without error into an executable:

```

algorithm(1026) : warning C4996: 'std::_Transform1': Function call with parameters
that may be unsafe - this call relies on the caller to check that the passed values
are correct. To disable this warning, use -D_SCL_SECURE_NO_WARNINGS. See documentation
on how to use Visual C++ 'Checked Iterators'

```

When run at the command line, the executable generates this output:

```

v: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
v2: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
v3: 0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
a4: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
a5: 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
a6: 0 6 12 18 24 30 36 42 48 54 60 66 72 78 84 90
a7: 0 7 14 21 28 35 42 49 56 63 70 77 84 91 98 105
a8: 0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120

```

See also

[C++ Standard Library Overview](#)

[Debug Iterator Support](#)

_SECURE_SCL

10/31/2018 • 2 minutes to read • [Edit Online](#)

Superseded by [_ITERATOR_DEBUG_LEVEL](#), this macro defines whether [Checked Iterators](#) are enabled. By default, checked iterators are enabled in Debug builds, and disabled in Retail builds.

IMPORTANT

Direct use of the `_SECURE_SCL` macro is deprecated. Instead, use `_ITERATOR_DEBUG_LEVEL` to control checked iterator settings. For more information, see [_ITERATOR_DEBUG_LEVEL](#).

Remarks

When checked iterators are enabled, unsafe iterator use causes a runtime error and the program is terminated. To enable checked iterators, set `_ITERATOR_DEBUG_LEVEL` to 1 or 2. This is equivalent to a `_SECURE_SCL` setting of 1, or enabled:

```
#define _ITERATOR_DEBUG_LEVEL 1
```

To disable checked iterators, set `_ITERATOR_DEBUG_LEVEL` to 0. This is equivalent to a `_SECURE_SCL` setting of 0, or disabled:

```
#define _ITERATOR_DEBUG_LEVEL 0
```

For information on how to disable warnings about checked iterators, see [_SCL_SECURE_NO_WARNINGS](#).

See also

[_ITERATOR_DEBUG_LEVEL](#)

[Checked Iterators](#)

[Debug Iterator Support](#)

[Safe Libraries: C++ Standard Library](#)

Debug Iterator Support

3/21/2019 • 3 minutes to read • [Edit Online](#)

The Visual C++ run-time library detects incorrect iterator use, and asserts and displays a dialog box at run time. To enable debug iterator support, you must use debug versions of the C++ Standard Library and C Runtime Library to compile your program. For more information, see [CRT Library Features](#). For information about how to use checked iterators, see [Checked Iterators](#).

The C++ standard describes how member functions might cause iterators to a container to become invalid. Two examples are:

- Erasing an element from a container causes iterators to the element to become invalid.
- Increasing the size of a [vector](#) by using push or insert causes iterators into the `vector` to become invalid.

Invalid iterators

If you compile this sample program in debug mode, at run time it asserts and terminates.

```
// iterator_debugging_0.cpp
// compile by using /EHsc /MDd
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v {10, 15, 20};
    std::vector<int>::iterator i = v.begin();
    ++i;

    std::vector<int>::iterator j = v.end();
    --j;

    std::cout << *j << '\n';

    v.insert(i,25);

    std::cout << *j << '\n'; // Using an old iterator after an insert
}
```

Using _ITERATOR_DEBUG_LEVEL

You can use the preprocessor macro `_ITERATOR_DEBUG_LEVEL` to turn off the iterator debugging feature in a debug build. This program does not assert, but still triggers undefined behavior.

```
// iterator_debugging_1.cpp
// compile by using: /EHsc /MDd
#define _ITERATOR_DEBUG_LEVEL 0
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v {10, 15, 20};

    std::vector<int>::iterator i = v.begin();
    ++i;

    std::vector<int>::iterator j = v.end();
    --j;

    std::cout << *j << '\n';

    v.insert(i,25);

    std::cout << *j << '\n'; // Using an old iterator after an insert
}
```

```
20
-572662307
```

Unitialized iterators

An assert also occurs if you attempt to use an iterator before it is initialized, as shown here:

```
// iterator_debugging_2.cpp
// compile by using: /EHsc /MDd
#include <string>
using namespace std;

int main() {
    string::iterator i1, i2;
    if (i1 == i2)
        ;
}
```

Incompatible iterators

The following code example causes an assertion because the two iterators to the [for_each](#) algorithm are incompatible. Algorithms check to determine whether the iterators that are supplied to them reference the same container.

```

// iterator_debugging_3.cpp
// compile by using /EHsc /MDd
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1 {10, 20};
    vector<int> v2 {10, 20};

    // The next line asserts because v1 and v2 are
    // incompatible.
    for_each(v1.begin(), v2.end(), [] (int& elem) { elem *= 2; } );
}

```

Notice that this example uses the lambda expression `[] (int& elem) { elem *= 2; }` instead of a functor. Although this choice has no bearing on the assert failure—a similar functor would cause the same failure—lambdas are a very useful way to accomplish compact function object tasks. For more information about lambda expressions, see [Lambda Expressions](#).

Iterators going out of scope

Debug iterator checks also cause an iterator variable that's declared in a **for** loop to be out of scope when the **for** loop scope ends.

```

// iterator_debugging_4.cpp
// compile by using: /EHsc /MDd
#include <vector>
#include <iostream>
int main() {
    std::vector<int> v {10, 15, 20};

    for (std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
        ; // do nothing
    --i; // C2065
}

```

Destructors for debug iterators

Debug iterators have non-trivial destructors. If a destructor does not run but the object's memory is freed, access violations and data corruption might occur. Consider this example:

```
// iterator_debugging_5.cpp
// compile by using: /EHsc /MDd
#include <vector>
struct base {
    // TO FIX: uncomment the next line
    // virtual ~base() {}
};

struct derived : base {
    std::vector<int>::iterator m_iter;
    derived( std::vector<int>::iterator iter ) : m_iter( iter ) {}
    ~derived() {}
};

int main() {
    std::vector<int> vect( 10 );
    base * pb = new derived( vect.begin() );
    delete pb; // doesn't call ~derived()
               // access violation
}
```

See also

[C++ Standard Library Overview](#)

_HAS_ITERATOR_DEBUGGING

10/31/2018 • 2 minutes to read • [Edit Online](#)

Superseded by [_ITERATOR_DEBUG_LEVEL](#), this macro defines whether the iterator debugging feature is enabled in a debug build. By default, iterator debugging is enabled in Debug builds and disabled in Retail builds. For more information, see [Debug Iterator Support](#).

IMPORTANT

Direct use of the `_HAS_ITERATOR_DEBUGGING` macro is deprecated. Instead, use `_ITERATOR_DEBUG_LEVEL` to control iterator debug settings. For more information, see [_ITERATOR_DEBUG_LEVEL](#).

Remarks

To enable iterator debugging in debug builds, set `_ITERATOR_DEBUG_LEVEL` to 2. This is equivalent to a `_HAS_ITERATOR_DEBUGGING` setting of 1, or enabled:

```
#define _ITERATOR_DEBUG_LEVEL 2
```

`_ITERATOR_DEBUG_LEVEL` cannot be set to 2 (and `_HAS_ITERATOR_DEBUGGING` cannot be set to 1) in retail builds.

To disable debug iterators in debug builds, set `_ITERATOR_DEBUG_LEVEL` to 0 or 1. This is equivalent to a `_HAS_ITERATOR_DEBUGGING` setting of 0, or disabled:

```
#define _ITERATOR_DEBUG_LEVEL 0
```

See also

[_ITERATOR_DEBUG_LEVEL](#)

[Debug Iterator Support](#)

[Checked Iterators](#)

[Safe Libraries: C++ Standard Library](#)

Thread Safety in the C++ Standard Library

10/31/2018 • 2 minutes to read • [Edit Online](#)

The following thread safety rules apply to all classes in the C++ Standard Library—this includes `shared_ptr`, as described below. Stronger guarantees are sometimes provided—for example, the standard iostream objects, as described below, and types specifically intended for multithreading, like those in `<atomic>`.

An object is thread-safe for reading from multiple threads. For example, given an object A, it is safe to read A from thread 1 and from thread 2 simultaneously.

If an object is being written to by one thread, then all reads and writes to that object on the same or other threads must be protected. For example, given an object A, if thread 1 is writing to A, then thread 2 must be prevented from reading from or writing to A.

It is safe to read and write to one instance of a type even if another thread is reading or writing to a different instance of the same type. For example, given objects A and B of the same type, it is safe when A is being written in thread 1 and B is being read in thread 2.

shared_ptr

Multiple threads can simultaneously read and write different `shared_ptr` objects, even when the objects are copies that share ownership.

iostream

The standard iostream objects `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr`, and `wclog` follow the same rules as the other classes, with this exception: it is safe to write to an object from multiple threads. For example, thread 1 can write to `cout` at the same time as thread 2. However, this can cause the output from the two threads to be intermixed.

NOTE

Reading from a stream buffer is not considered to be a read operation. Instead it is considered to be a write operation because the state of the class is changed.

See also

[C++ Standard Library Overview](#)

stdext Namespace

3/11/2019 • 2 minutes to read • [Edit Online](#)

Members of the `<hash_map>` and `<hash_set>` header files are not currently part of the ISO C++ standard. Therefore, these types and members have been moved from the `std` namespace to namespace `stdext`, to remain conformant with the C++ standard.

When compiling with `/Ze`, which is the default, the compiler warns on the use of `std` for members of the `<hash_map>` and `<hash_set>` header files. To disable the warning, use the `warning` pragma.

To have the compiler generate an error for the use of `std` for members of the `<hash_map>` and `<hash_set>` header files with `/Ze`, add the following directive before you `#include` any C++ Standard Library header files.

```
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
```

When compiling with `/Za`, the compiler generates an error.

See also

[C++ Standard Library Overview](#)

C++ Standard Library Containers

11/9/2018 • 7 minutes to read • [Edit Online](#)

The Standard Library provides various type-safe containers for storing collections of related objects. The containers are class templates; when you declare a container variable, you specify the type of the elements that the container will hold. Containers can be constructed with initializer lists. They have member functions for adding and removing elements and performing other operations.

You iterate over the elements in a container, and access the individual elements by using [iterators](#). You can use iterators explicitly by using their member functions and operators as well as global functions. You can also use them implicitly, for example by using a range-for loop. Iterators for all C++ Standard Library containers have a common interface but each container defines its own specialized iterators.

Containers can be divided into three categories: sequence containers, associative containers, and container adapters.

Sequence Containers

Sequence containers maintain the ordering of inserted elements that you specify.

A `vector` container behaves like an array, but can automatically grow as required. It is random access and contiguously stored, and length is highly flexible. For these reasons and more, `vector` is the preferred sequence container for most applications. When in doubt as to what kind of sequence container to use, start by using a vector! For more information, see [vector Class](#).

An `array` container has some of the strengths of `vector`, but the length is not as flexible. For more information, see [array Class](#).

A `deque` (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. It shares the random-access and flexible-length advantages of `vector`, but is not contiguous. For more information, see [deque Class](#).

A `list` container is a doubly linked list that enables bidirectional access, fast insertions, and fast deletions anywhere in the container, but you cannot randomly access an element in the container. For more information, see [list Class](#).

A `forward_list` container is a singly linked list—the forward-access version of `list`. For more information, see [forward_list Class](#).

Associative Containers

In associative containers, elements are inserted in a pre-defined order—for example, as sorted ascending. Unordered associative containers are also available. The associative containers can be grouped into two subsets: maps and sets.

A `map`, sometimes referred to as a dictionary, consists of a key/value pair. The key is used to order the sequence, and the value is associated with that key. For example, a `map` might contain keys that represent every unique word in a text and corresponding values that represent the number of times that each word appears in the text. The unordered version of `map` is `unordered_map`. For more information, see [map Class](#) and [unordered_map Class](#).

A `set` is just an ascending container of unique elements—the value is also the key. The unordered version of

`set` is `unordered_set`. For more information, see [set Class](#) and [unordered_set Class](#).

Both `map` and `set` only allow one instance of a key or element to be inserted into the container. If multiple instances of elements are required, use `multimap` or `multiset`. The unordered versions are `unordered_multimap` and `unordered_multiset`. For more information, see [multimap Class](#), [unordered_multimap Class](#), [multiset Class](#), and [unordered_multiset Class](#).

Ordered maps and sets support bi-directional iterators, and their unordered counterparts support forward iterators. For more information, see [Iterators](#).

Heterogeneous Lookup in Associative Containers (C++14)

The ordered associative containers (`map`, `multimap`, `set` and `multiset`) now support heterogeneous lookup, which means that you are no longer required to pass the exact same object type as the key or element in member functions such as `find()` and `lower_bound()`. Instead, you can pass any type for which an overloaded `operator<` is defined that enables comparison to the key type.

Heterogeneous lookup is enabled on an opt-in basis when you specify the `std::less<>` or `std::greater<>` "diamond functor" comparator when declaring the container variable, as shown here:

```
std::set<BigObject, std::less<>> myNewSet;
```

If you use the default comparator, then the container behaves exactly as it did in C++11 and earlier.

The following example shows how to overload `operator<` in order to enable users of a `std::set` to do lookups simply by passing in a small string that can be compared to each object's `BigObject::id` member.

```

#include <set>
#include <string>
#include <iostream>
#include <functional>

using namespace std;

class BigObject
{
public:
    string id;
    explicit BigObject(const string& s) : id(s) {}
    bool operator< (const BigObject& other) const
    {
        return this->id < other.id;
    }

    // Other members....
};

inline bool operator<(const string& otherId, const BigObject& obj)
{
    return otherId < obj.id;
}

inline bool operator<(const BigObject& obj, const string& otherId)
{
    return obj.id < otherId;
}

int main()
{
    // Use C++14 brace-init syntax to invoke BigObject(string).
    // The s suffix invokes string ctor. It is a C++14 user-defined
    // literal defined in <string>
    BigObject b1{ "42F"s };
    BigObject b2{ "52F"s };
    BigObject b3{ "62F"s };
    set<BigObject, less<>> myNewSet; // C++14
    myNewSet.insert(b1);
    myNewSet.insert(b2);
    myNewSet.insert(b3);
    auto it = myNewSet.find(string("62F"));
    if (it != myNewSet.end())
        cout << "myNewSet element = " << it->id << endl;
    else
        cout << "element not found " << endl;

    // Keep console open in debug mode:
    cout << endl << "Press Enter to exit.";
    string s;
    getline(cin, s);
    return 0;
}

//Output: myNewSet element = 62F

```

The following member functions in map, multimap, set and multiset have been overloaded to support heterogeneous lookup:

1. find
2. count
3. lower_bound

4. `upper_bound`

5. `equal_range`

Container Adapters

A container adapter is a variation of a sequence or associative container that restricts the interface for simplicity and clarity. Container adapters do not support iterators.

A `queue` container follows FIFO (first in, first out) semantics. The first element *pushed*—that is, inserted into the queue—is the first to be *popped*—that is, removed from the queue. For more information, see [queue Class](#).

A `priority_queue` container is organized such that the element that has the highest value is always first in the queue. For more information, see [priority_queue Class](#).

A `stack` container follows LIFO (last in, first out) semantics. The last element pushed on the stack is the first element popped. For more information, see [stack Class](#).

Because container adapters do not support iterators, they cannot be used with the C++ Standard Library algorithms. For more information, see [Algorithms](#).

Requirements for Container Elements

In general, elements inserted into a C++ Standard Library container can be of just about any object type if they are copyable. Movable-only elements—for example, those such as `vector<unique_ptr<T>>` that are created by using `unique_ptr<>` will work as long as you don't call member functions that attempt to copy them.

The destructor is not permitted to throw an exception.

Ordered associative containers—described earlier in this article—must have a public comparison operator defined. (By default, the operator is `operator<`, but even types that don't work with `operator<` are supported.

Some operations on containers might also require a public default constructor and a public equivalence operator. For example, the unordered associative containers require support for equality and hashing.

Accessing Container Elements

The elements of containers are accessed by using iterators. For more information, see [Iterators](#).

NOTE

You can also use [range-based for loops](#) to iterate over C++ Standard Library collections.

Comparing containers

All containers overload the `operator==` for comparing two containers of the same type that have the same element type. You can use `==` to compare a `vector<string>` to another `vector<string>`, but you cannot use it to compare a `vector<string>` to a `list<string>` or a `vector<string>` to a `vector<char*>`. In C++98/03 you can use [std::equal](#) or [std::mismatch](#) to compare dissimilar container types and/or element types. In C++11 you can also use [std::is_permutation](#). But in all these cases the functions assume that the containers are the same length. If the second range is shorter than the first, then undefined behavior results. If the second range is longer, results can still be incorrect because the comparison never continues past the end of the first range.

Comparing dissimilar containers (C++14)

In C++14 and later, you can compare dissimilar containers and/or dissimilar elements types by using one of the `std::equal`, `std::mismatch`, or `std::is_permutation` function overloads that take two complete ranges.

These overloads enable you to compare containers with different lengths. These overloads are much less susceptible to user error, and are optimized to return false in constant time when containers of dissimilar lengths are compared. Therefore, we recommend you use these overloads unless (1) you have a very clear reason not to, or (2) you are using a [std::list](#) container, which does not benefit from the dual-range optimizations.

See also

[Containers](#)

[C++ Standard Library Reference](#)

[<sample container>](#)

[Thread Safety in the C++ Standard Library](#)

<sample container>

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Shows you the structure of the container headers in the C++ Standard Library.

Syntax

```
#include <sample container> // Nonfunctional header
```

<sample container> Operators

3/11/2019 • 2 minutes to read • [Edit Online](#)

For more information about the operators in <sample container>, see [<sample container>](#).

operator!=

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads `operator!=` to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator!=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(left == right)`.

See also

[<sample container>](#)

operator== (<sample container>)

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads `operator==` to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator==(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `left.size == right.size && equal(left.begin, left.end, right.begin)`.

See also

[<sample container>](#)

operator< (<sample container>)

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads **operator<** to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator<(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `lexicographical_compare(left.begin, left.end, right.begin, right.end)`.

See also

[<sample container>](#)

[begin](#)

[end](#)

operator<= (<sample container>)

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads **operator<=** to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator<=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(right < left)`.

See also

[<sample container>](#)

operator> (<sample container>)

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads **operator>** to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator*>(<
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `right < left`.

See also

[<sample container>](#)

operator>=

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Overloads **operator>=** to compare two objects of template class [Container](#).

Syntax

```
template <class Ty>
bool operator>=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(left < right)`.

See also

[<sample container>](#)

<sample container> Specialized Template Functions

3/11/2019 • 2 minutes to read • [Edit Online](#)

For more information about the specialized template functions in <sample container>, see [<sample container>](#).

swap (<sample container>)

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Executes `left.swap(right)`.

Syntax

```
template <class Ty>
void swap(
    Container <Ty>& left,
    Container <Ty>& right);
```

See also

[<sample container>](#)

<sample container> Classes

3/11/2019 • 2 minutes to read • [Edit Online](#)

For more information about the classes in <sample container>, see [<sample container>](#).

Sample Container Class

5/7/2019 • 3 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that controls a varying-length sequence of elements, typically of type `Ty`. The sequence is stored in different ways, depending on the actual container.

A container constructor or member function may find occasion to call the constructor **`Ty(const Ty&)`** or the function **`Ty::operator=(const Ty&)`**. If such a call throws an exception, the container object is obliged to maintain its integrity, and to rethrow any exception it catches. You can safely swap, assign to, erase, or destroy a container object after it throws one of these exceptions. In general, however, you cannot otherwise predict the state of the sequence controlled by the container object.

A few additional caveats:

- If the expression `~Ty` throws an exception, the resulting state of the container object is undefined.
- If the container stores an allocator object `al`, and `al` throws an exception other than as a result of a call to `al.allocate`, the resulting state of the container object is undefined.
- If the container stores a function object `comp`, to determine how to order the controlled sequence, and `comp` throws an exception of any kind, the resulting state of the container object is undefined.

The container classes defined by C++ Standard Library satisfy several additional requirements, as described in the following paragraphs.

Container template class `list` provides deterministic, and useful, behavior even in the presence of the exceptions described above. For example, if an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by C++ Standard Library, if an exception is thrown during calls to the following member functions, `insert`, `push_back`, or `push_front`, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by C++ Standard Library, no exception is thrown during calls to the following member functions: `pop_back`, `pop_front`.

The member function `erase` throws an exception only if a copy operation (assignment or copy construction) throws an exception.

Moreover, no exception is thrown while copying an iterator returned by a member function.

The member function `swap` makes additional promises for *all* container classes defined by C++ Standard Library:

- The member function throws an exception only if the container stores an allocator object `al`, and `al` throws an exception when copied.
- References, pointers, and iterators that designate elements of the controlled sequences being swapped remain valid.

An object of a container class defined by C++ Standard Library allocates and frees storage for the sequence it controls through a stored object of type `Alloc`, which is typically a template parameter. Such an allocator object must have the same external interface as an object of class `allocator<Ty>`. In particular, `Alloc` must be the same type as `Alloc::rebind<value_type>::other`.

For *all* container classes defined by C++ Standard Library, the member function `Alloc get_allocator const;` returns a copy of the stored allocator object. Note that the stored allocator object is *not* copied when the container object is assigned. All constructors initialize the value stored in `allocator`, to `Alloc` if the constructor contains no allocator parameter.

According to the C++ Standard, a container class defined by the C++ Standard Library can assume that:

- All objects of class `Alloc` compare equal.
- Type `Alloc::const_pointer` is the same as `const Ty *`.
- Type `Alloc::const_reference` is the same as `const Ty&`.
- Type `Alloc::pointer` is the same as `Ty *`.
- Type `Alloc::reference` is the same as `Ty&`.

In this implementation, however, containers do not make such simplifying assumptions. Thus, they work properly with allocator objects that are more ambitious:

- All objects of class `Alloc` does not need to compare equal. (You can maintain multiple pools of storage.)
- Type `Alloc::const_pointer` does not need to be the same as `const Ty *`. (A const pointer can be a class.)
- Type `Alloc::pointer` does not need to be the same as `Ty *`. (A pointer can be a class.)

Requirements

Header: `<sample container>`

See also

[<sample container>](#)

Sample Container Members

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Reference

Typedefs

const_iterator	Describes an object that can serve as a constant iterator for the controlled sequence.
const_reference	Describes an object that can serve as a constant reference to an element of the controlled sequence.
const_reverse_iterator	Describes an object that can serve as a constant reverse iterator for the controlled sequence.
difference_type	Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.
iterator	Describes an object that can serve as an iterator for the controlled sequence.
reference	Describes an object that can serve as a reference to an element of the controlled sequence.
reverse_iterator	Describes an object that can serve as a reverse iterator for the controlled sequence.
size_type	Describes an object that can represent the length of any controlled sequence.
value_type	Acts a synonym for the template parameter <code>Ty</code> .

Member Functions

begin	Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).
clear	Calls erase(begin, end) .
empty	Returns true for an empty controlled sequence.

--	--

end	Returns an iterator that points just beyond the end of the sequence.
erase	Erases an element.
max_size	Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.
rbegin	Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.
rend	The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.
size	Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.
swap	

Sample Container Typedefs

10/31/2018 • 2 minutes to read • [Edit Online](#)

For more information about the typedefs in the sample container class, see [Sample Container Class](#)

Container Class::const_iterator

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as a constant iterator for the controlled sequence.

Syntax

```
typedef T6 const_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T6`.

See also

[Sample Container Class](#)

Container Class::const_reference

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as a constant reference to an element of the controlled sequence.

Syntax

```
typedef T3 const_reference;
```

Remarks

It is described here as a synonym for the unspecified type `T3` (typically `Alloc::const_reference`).

See also

[Sample Container Class](#)

Container Class::const_reverse_iterator

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as a constant reverse iterator for the controlled sequence.

Syntax

```
typedef T8 const_reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T8` (typically `reverse_iterator <const_iterator >`).

See also

[Sample Container Class](#)

Container Class::difference_type

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

Syntax

```
typedef T1 difference_type;
```

Remarks

It is described here as a synonym for the unspecified type `T1` (typically `Alloc::difference_type`).

See also

[Sample Container Class](#)

Container Class::iterator

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as an iterator for the controlled sequence.

Syntax

```
typedef T5 iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T5`. An object of type `iterator` can be cast to an object of type [const_iterator](#).

See also

[Sample Container Class](#)

Container Class::reference

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as a reference to an element of the controlled sequence.

Syntax

```
typedef T2 reference;
```

Remarks

It is described here as a synonym for the unspecified type `T2` (typically `Alloc::reference`). An object of type `reference` can be cast to an object of type [const_reference](#).

See also

[Sample Container Class](#)

Container Class::reverse_iterator

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can serve as a reverse iterator for the controlled sequence.

Syntax

```
typedef T7 reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T7` (typically `reverse_iterator` `<iterator>`).

See also

[Sample Container Class](#)

Container Class::size_type

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Describes an object that can represent the length of any controlled sequence.

Syntax

```
typedef T0 size_type;
```

Remarks

It is described here as a synonym for the unspecified type `T0` (typically `Alloc::size_type`).

See also

[Sample Container Class](#)

Container Class::value_type

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Acts a synonym for the template parameter *Ty*.

Syntax

```
typedef T4 value_type;
```

Remarks

It is described here as a synonym for the unspecified type `T4` (typically `Alloc::value_type`).

See also

[Sample Container Class](#)

Sample Container Member Functions

10/31/2018 • 2 minutes to read • [Edit Online](#)

For more information about the member functions in the sample container class, see [Sample Container Class](#)

Container Class::begin

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Visual Studio C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Syntax

```
const_iterator begin() const;  
  
iterator begin();
```

See also

[Sample Container Class](#)

Container Class::clear

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Calls [erase\(begin, end\)](#).

Syntax

```
void clear();
```

See also

[Sample Container Class](#)

Container Class::empty

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns **true** for an empty controlled sequence.

Syntax

```
bool empty() const;
```

See also

[Sample Container Class](#)

Container Class::end

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns an iterator that points just beyond the end of the sequence.

Syntax

```
const_iterator end() const;  
  
iterator end();
```

See also

[Sample Container Class](#)

Container Class::erase

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Erases an element.

Syntax

```
iterator erase(  
    iterator _Where);  
  
iterator erase(  
    iterator first,  
    iterator last);
```

Remarks

The first member function removes the element of the controlled sequence pointed to by *_Where*. The second member function removes the elements of the controlled sequence in the range [*first*, *last*). Both return an iterator that designates the first element remaining beyond any elements removed, or [end](#) if no such element exists.

The member functions throw an exception only if a copy operation throws an exception.

See also

[Sample Container Class](#)

Container Class::max_size

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.

Syntax

```
size_type max_size() const;
```

See also

[Sample Container Class](#)

Container Class::rbegin

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.

Syntax

```
const_reverse_iterator rbegin() const;  
  
reverse_iterator rbegin();
```

See also

[Sample Container Class](#)

Container Class::rend

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.

Syntax

```
const_reverse_iterator rend() const;  
reverse_iterator rend();
```

See also

[Sample Container Class](#)

Container Class::size

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.

Syntax

```
size_type size() const;
```

See also

[Sample Container Class](#)

Container Class::swap

5/7/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see [C++ Standard Library Containers](#).

Swaps the controlled sequences between ***this** and its argument.

Syntax

```
void swap(Container& right);
```

Remarks

If ***this.get_allocator == right.get_allocator**, it does a swap in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

See also

[Sample Container Class](#)

Iterators

5/7/2019 • 4 minutes to read • [Edit Online](#)

An iterator is an object that can iterate over elements in a C++ Standard Library container and provide access to individual elements. The C++ Standard Library containers all provide iterators so that algorithms can access their elements in a standard way without having to be concerned with the type of container the elements are stored in.

You can use iterators explicitly using member and global functions such as `begin()` and `end()` and operators such as `++` and `--` to move forward or backward. You can also use iterators implicitly with a range-for loop or (for some iterator types) the subscript operator `[]`.

In the C++ Standard Library, the beginning of a sequence or range is the first element. The end of a sequence or range is always defined as one past the last element. The global functions `begin` and `end` return iterators to a specified container. The typical explicit iterator loop over all elements in a container looks like this:

```
vector<int> vec{ 0,1,2,3,4 };
for (auto it = begin(vec); it != end(vec); it++)
{
    // Access element using dereference operator
    cout << *it << " ";
}
```

The same thing can be accomplished more simply with a range-for loop:

```
for (auto num : vec)
{
    // no dereference operator
    cout << num << " ";
}
```

There are five categories of iterators. In order of increasing power, the categories are:

- **Output.** An *output iterator* `x` can iterate forward over a sequence by using the `++` operator, and can write an element only once, by using the `*` operator.
- **Input.** An *input iterator* `x` can iterate forward over a sequence by using the `++` operator, and can read an element any number of times by using the `*` operator. You can compare input iterators by using the `++` and `!=` operators. After you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- **Forward.** A *forward iterator* `x` can iterate forward over a sequence using the `++` operator and can read any element or write non-const elements any number of times by using the `*` operator. You can access element members by using the `->` operator and compare forward iterators by using the `==` and `!=` operators. You can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently. A forward iterator that is initialized without reference to any container is called a *null forward iterator*. Null forward iterators always compare equal.
- **Bidirectional.** A *bidirectional iterator* `x` can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in `--x`, `x--`, or `(v = *x--)`. You can access element members and compare bidirectional iterators in the same way as forward iterators.
- **Random access.** A *random-access iterator* `x` can take the place of a bidirectional iterator. With a random access iterator you can use the subscript operator `[]` to access elements. You can use the `+`, `-`, `+=` and `-=`

operators to move forward or backward a specified number of elements and to calculate the distance between iterators. You can compare bidirectional iterators by using `==`, `!=`, `<`, `>`, `<=`, and `>=`.

All iterators can be assigned or copied. They are assumed to be lightweight objects and are often passed and returned by value, not by reference. Note also that none of the operations previously described can throw an exception when performed on a valid iterator.

The hierarchy of iterator categories can be summarized by showing three sequences. For write-only access to a sequence, you can use any of:

```
output iterator
-> forward iterator
-> bidirectional iterator
-> random-access iterator
```

The right arrow means "can be replaced by." Any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

```
input iterator
-> forward iterator
-> bidirectional iterator
-> random-access iterator
```

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

```
forward iterator
-> bidirectional iterator
-> random-access iterator
```

An object pointer can always serve as a random-access iterator, so it can serve as any category of iterator if it supports the proper read/write access to the sequence it designates.

An iterator `Iterator` other than an object pointer must also define the member types required by the specialization `iterator_traits<Iterator>`. Note that these requirements can be met by deriving `Iterator` from the public base class `iterator`.

It is important to understand the promises and limitations of each iterator category to see how iterators are used by containers and algorithms in the C++ Standard Library.

NOTE

You can avoid using iterators explicitly by using range-for loops. For more information, see [Range-based for statement](#).

Microsoft C++ now offers checked iterators and debug iterators to ensure that you do not overwrite the bounds of your container. For more information, see [Checked Iterators](#) and [Debug Iterator Support](#).

See also

[C++ Standard Library Reference](#)

[Thread Safety in the C++ Standard Library](#)

Algorithms

10/31/2018 • 4 minutes to read • [Edit Online](#)

Algorithms are a fundamental part of the C++ Standard Library. Algorithms do not work with containers themselves but rather with iterators. Therefore, the same algorithm can be used by most if not all of the C++ Standard Library containers. This section discusses the conventions and terminology of the C++ Standard Library algorithms.

Remarks

The descriptions of the algorithm template functions employ several shorthand phrases:

- The phrase "in the range $[A, B)$ " means the sequence of zero or more discrete values beginning with A up to but not including B . A range is valid only if B is reachable from A ; you can store A in an object N ($N = A$), increment the object zero or more times ($++N$), and have the object compare equal to B after a finite number of increments ($N == B$).
- The phrase "each N in the range $[A, B)$ " means that N begins with the value A and is incremented zero or more times until it equals the value B . The case $N == B$ is not in the range.
- The phrase "the lowest value of N in the range $[A, B)$ such that X " means that the condition X is determined for each N in the range $[A, B)$ until the condition X is met.
- The phrase "the highest value of N in the range $[A, B)$ such that X " means that X is determined for each N in the range $[A, B)$. The function stores in K a copy of N each time the condition X is met. If any such store occurs, the function replaces the final value of N , which equals B , with the value of K . For a bidirectional or random-access iterator, however, it can also mean that N begins with the highest value in the range and is decremented over the range until the condition X is met.
- Expressions such as $X - Y$, where X and Y can be iterators other than random-access iterators, are intended in the mathematical sense. The function does not necessarily evaluate operator $-$ if it must determine such a value. The same is also true for expressions such as $X + N$ and $X - N$, where N is an integer type.

Several algorithms make use of a predicate that performs a pairwise comparison, such as with `operator==`, to yield a **bool** result. The predicate function `operator==`, or any replacement for it, must not alter either of its operands. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

Several algorithms make use of a predicate that must impose a strict weak ordering on pairs of elements from a sequence. For the predicate $pred(X, Y)$:

- Strict means that $pred(X, X)$ is false.
- Weak means that X and Y have an equivalent ordering if $!pred(X, Y) \ \&\& \ !pred(Y, X)$ ($X == Y$ does not need to be defined).
- Ordering means that $pred(X, Y) \ \&\& \ pred(Y, Z)$ implies $pred(X, Z)$.

Some of these algorithms implicitly use the predicate $X < Y$. Other predicates that typically satisfy the strict weak ordering requirement are $X > Y$, `less`(X, Y), and `greater`(X, Y). Note, however, that predicates such as $X <= Y$ and $X >= Y$ do not satisfy this requirement.

A sequence of elements designated by iterators in the range $[First, Last)$ is a sequence ordered by operator $<$ if, for each N in the range $[0, Last - First)$ and for each M in the range $(N, Last - First)$ the predicate $!(*(First + M) < *(First + N))$ is false.

+ N) is true. (Note that the elements are sorted in ascending order.) The predicate function `operator<`, or any replacement for it, must not alter either of its operands. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand. Moreover, it must impose a strict weak ordering on the operands it compares.

A sequence of elements designated by iterators in the range `[First, Last)` is a heap ordered by `operator<` if, for each N in the range `[1, Last - First)` the predicate `!(*First < *(First + N))` is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions `make_heap`, `pop_heap`, and `push_heap`. As with an ordered sequence, the predicate function `operator<`, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering on the operands it compares. It must yield the same **bool** result every time it is evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

The C++ Standard Library algorithms are located in the `<algorithm>` and `<numeric>` header files.

See also

[C++ Standard Library Reference](#)

[Thread Safety in the C++ Standard Library](#)

Allocators

10/31/2018 • 3 minutes to read • [Edit Online](#)

Allocators are used by the C++ Standard Library to handle the allocation and deallocation of elements stored in containers. All C++ Standard Library containers except `std::array` have a template parameter of type `allocator<Type>`, where `Type` represents the type of the container element. For example, the vector class is declared as follows:

```
template <
    class Type,
    class Allocator = allocator<Type>
>
class vector
```

The C++ Standard Library provides a default implementation for an allocator. In C++11 and later, the default allocator is updated to expose a smaller interface; the new allocator is called a *minimal allocator*. In particular, the minimal allocator's `construct()` member supports move semantics, which can greatly improve performance. In most cases, this default allocator should be sufficient. In C++11 all the Standard Library types and functions that take an allocator type parameter support the minimal allocator interface, including `std::function`, `shared_ptr`, `allocate_shared()`, and `basic_string`. For more information on the default allocator, see [allocator Class](#).

Writing Your Own Allocator (C++11)

The default allocator uses **new** and **delete** to allocate and deallocate memory. If you want to use a different method of memory allocation, such as using shared memory, then you must create your own allocator. If you are targeting C++11 and you need to write a new custom allocator, make it a minimal allocator if possible. Even if you have already implemented an old-style allocator, consider modifying it to be a *minimal allocator* in order to take advantage of the more efficient `construct()` method that will be provided for you automatically.

A minimal allocator requires much less boilerplate and enable you to focus on the `allocate` and `deallocate` member functions which do all of the work. When creating a minimal allocator, do not implement any members except the ones shown in the example below:

1. a converting copy constructor (see example)
2. `operator==`
3. `operator!=`
4. `allocate`
5. `deallocate`

The C++11 default `construct()` member that will be provided for you does perfect forwarding and enables move semantics; it is much more efficient in many cases than the older version.

WARNING

At compile time, the C++ Standard Library uses the `allocator_traits` class to detect which members you have explicitly provided and provides a default implementation for any members that are not present. Do not interfere with this mechanism by providing a specialization of `allocator_traits` for your allocator!

The following example shows a minimal implementation of an allocator that uses `malloc` and `free`. Note the use of the new exception type `std::bad_array_new_length` which is thrown if the array size is less than zero or greater than the maximum allowed size.

```
#pragma once
#include <stdlib.h> //size_t, malloc, free
#include <new> // bad_alloc, bad_array_new_length
#include <memory>
template <class T>
struct Mallocator
{
    typedef T value_type;
    Mallocator() noexcept {} //default ctor not required by C++ Standard Library

    // A converting copy constructor:
    template<class U> Mallocator(const Mallocator<U>&) noexcept {}
    template<class U> bool operator==(const Mallocator<U>&) const noexcept
    {
        return true;
    }
    template<class U> bool operator!=(const Mallocator<U>&) const noexcept
    {
        return false;
    }
    T* allocate(const size_t n) const;
    void deallocate(T* const p, size_t) const noexcept;
};

template <class T>
T* Mallocator<T>::allocate(const size_t n) const
{
    if (n == 0)
    {
        return nullptr;
    }
    if (n > static_cast<size_t>(-1) / sizeof(T))
    {
        throw std::bad_array_new_length();
    }
    void* const pv = malloc(n * sizeof(T));
    if (!pv) { throw std::bad_alloc(); }
    return static_cast<T*>(pv);
}

template<class T>
void Mallocator<T>::deallocate(T * const p, size_t) const noexcept
{
    free(p);
}
```

Writing Your Own Allocator (C++03)

In C++03, any allocator used with C++ Standard Library containers must implement the following type definitions:

<code>const_pointer</code>	<code>rebind</code>
<code>const_reference</code>	<code>reference</code>
<code>difference_type</code>	<code>size_type</code>
<code>pointer</code>	<code>value_type</code>

In addition, any allocator used with C++ Standard Library containers must implement the following methods:

Constructor	<code>deallocate</code>
Copy constructor	<code>destroy</code>
Destructor	<code>max_size</code>
<code>address</code>	<code>operator==</code>
<code>allocate</code>	<code>operator!=</code>
<code>construct</code>	

For more information on these type definitions and methods, see [allocator Class](#).

See also

[C++ Standard Library Reference](#)

Function Objects in the C++ Standard Library

3/18/2019 • 2 minutes to read • [Edit Online](#)

A *function object*, or *functor*, is any type that implements `operator()`. This operator is referred to as the *call operator* or sometimes the *application operator*. The C++ Standard Library uses function objects primarily as sorting criteria for containers and in algorithms.

Function objects provide two main advantages over a straight function call. The first is that a function object can contain state. The second is that a function object is a type and therefore can be used as a template parameter.

Creating a Function Object

To create a function object, create a type and implement `operator()`, such as:

```
class Functor
{
public:
    int operator()(int a, int b)
    {
        return a < b;
    }
};

int main()
{
    Functor f;
    int a = 5;
    int b = 7;
    int ans = f(a, b);
}
```

The last line of the `main` function shows how you call the function object. This call looks like a call to a function, but it's actually calling `operator()` of the `Functor` type. This similarity between calling a function object and a function is how the term function object came about.

Function Objects and Containers

The C++ Standard Library contains several function objects in the `<functional>` header file. One use of these function objects is as a sorting criterion for containers. For example, the `set` container is declared as follows:

```
template <class Key,
          class Traits=less<Key>,
          class Allocator=allocator<Key>>
class set
```

The second template argument is the function object `less`. This function object returns **true** if the first parameter is less than the second parameter. Since some containers sort their elements, the container needs a way of comparing two elements. The comparison is done by using the function object. You can define your own sorting criteria for containers by creating a function object and specifying it in the template list for the container.

Function Objects and Algorithms

Another use of functional objects is in algorithms. For example, the `remove_if` algorithm is declared as follows:

```
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(
    ForwardIterator first,
    ForwardIterator last,
    Predicate pred);
```

The last argument to `remove_if` is a function object that returns a boolean value (a *predicate*). If the result of the function object is **true**, then the element is removed from the container being accessed by the iterators `first` and `last`. You can use any of the function objects declared in the [<functional>](#) header for the argument `pred` or you can create your own.

See also

[C++ Standard Library Reference](#)

iostream Programming

10/31/2018 • 2 minutes to read • [Edit Online](#)

This section provides a [general description](#) of the iostream classes and then describes [output streams](#), [input streams](#), and [input/output streams](#). The end of the section provides information about advanced iostream programming.

There is also a discussion on [Thread Safety in the C++ Standard Library](#) and [the stdext namespace](#).

In This Section

[What a Stream Is](#)

[Output Streams](#)

[Input Streams](#)

[Input/Output Streams](#)

[Custom Manipulators with Arguments](#)

See also

[C++ Standard Library](#)

[iostreams Conventions](#)

What a Stream Is

10/31/2018 • 2 minutes to read • [Edit Online](#)

Like C, C++ does not have built-in input/output capability. All C++ compilers, however, come bundled with a systematic, object-oriented I/O package, known as the `iostream` classes. The stream is the central concept of the `iostream` classes. You can think of a stream object as a smart file that acts as a source and destination for bytes. A stream's characteristics are determined by its class and by customized insertion and extraction operators.

Through device drivers, the disk operating system deals with the keyboard, screen, printer, and communication ports as extended files. The `iostream` classes interact with these extended files. Built-in classes support reading from and writing to memory with syntax identical to that for disk I/O, which makes it easy to derive stream classes.

In This Section

[Input/Output Alternatives](#)

See also

[iostream Programming](#)

Input/Output Alternatives

5/7/2019 • 2 minutes to read • [Edit Online](#)

The Microsoft C++ compiler provides several alternatives for I/O programming:

- C run-time library direct, unbuffered I/O.
- ANSI C run-time library stream I/O.
- Console and port direct I/O.
- Microsoft Foundation Class Library.
- Microsoft C++ Standard Library.

The `iostream` classes are useful for buffered, formatted text I/O. They are also useful for unbuffered or binary I/O if you need a C++ programming interface and decide not to use the Microsoft Foundation Class (MFC) library. The `iostream` classes are an object-oriented I/O alternative to the C run-time functions.

You can use `iostream` classes with the Microsoft Windows operating system. String and file streams work without restrictions, but the character-mode stream objects `cin`, `cout`, `cerr`, and `clog` are inconsistent with the Windows graphical user interface. You can also derive custom stream classes that interact directly with the Windows environment.

See also

[What a Stream Is](#)

Output Streams

10/31/2018 • 2 minutes to read • [Edit Online](#)

An output stream object is a destination for bytes. The three most important output stream classes are `ostream`, `ofstream`, and `ostringstream`.

The `ostream` class, through the derived class `basic_ostream`, supports the predefined stream objects:

- `cout` standard output
- `cerr` standard error with limited buffering
- `clog` similar to `cerr` but with full buffering

Objects are rarely constructed from `ostream`; predefined objects are generally used. In some cases, you can reassign predefined objects after program startup. The `ostream` class, which can be configured for buffered or unbuffered operation, is best suited to sequential text-mode output. All functionality of the base class, `ios`, is included in `ostream`. If you construct an object of class `ostream`, you must specify a `streambuf` object to the constructor.

The `ofstream` class supports disk file output. If you need an output-only disk, construct an object of class `ofstream`. You can specify whether `ofstream` objects accept binary or text-mode data when constructing the `ofstream` object or when calling the `open` member function of the object. Many formatting options and member functions apply to `ofstream` objects, and all functionality of the base classes `ios` and `ostream` is included.

If you specify a filename in the constructor, that file is automatically opened when the object is constructed. Otherwise, you can use the `open` member function after invoking the default constructor.

Like the run-time function `sprintf_s`, the `ostringstream` class supports output to in-memory strings. To create a string in memory by using I/O stream formatting, construct an object of class `ostringstream`.

In This Section

[Constructing Output Stream Objects](#)

[Using Insertion Operators and Controlling Format](#)

[Output File Stream Member Functions](#)

[Effects of Buffering](#)

[Binary Output Files](#)

[Overloading the << Operator for Your Own Classes](#)

[Writing Your Own Manipulators Without Arguments](#)

See also

[ofstream](#)

[ostringstream](#)

[iostream Programming](#)

Constructing Output Stream Objects

10/31/2018 • 2 minutes to read • [Edit Online](#)

If you use only the predefined `cout`, `cerr`, or `clog` objects, you do not need to construct an output stream. You must use constructors for:

- [Output File Stream Constructors](#)
- [Output String Stream Constructors](#)

Output File Stream Constructors

You can construct an output file stream in one of two ways:

- Use the default constructor, and then call the `open` member function.

```
ofstream myFile; // Static or on the stack
myFile.open("filename");

ofstream* pmyFile = new ofstream; // On the heap
pmyFile->open("filename");
```

- Specify a filename and mode flags in the constructor call.

```
ofstream myFile("filename", ios_base::out);
```

Output String Stream Constructors

To construct an output string stream, you can use `ostringstream` in the following way:

```
using namespace std;
// ...
ostringstream myString;
myString << "this is a test" << ends;

string sp = myString.str(); // Obtain string
cout << sp << endl;
```

The `ends` "manipulator" adds the necessary terminating null character to the string.

See also

[Output Streams](#)

Using Insertion Operators and Controlling Format

10/31/2018 • 5 minutes to read • [Edit Online](#)

This topic shows how to control format and how to create insertion operators for your own classes. The insertion (<<) operator, which is preprogrammed for all standard C++ data types, sends bytes to an output stream object. Insertion operators work with predefined "manipulators," which are elements that change the default format of integer arguments.

You can control the format with the following options:

- [Output Width](#)
- [Alignment](#)
- [Precision](#)
- [Radix](#)

Output Width

To align output, you specify the output width for each item by placing the `setw` manipulator in the stream or by calling the `width` member function. This example right-aligns the values in a column at least 10 characters wide:

```
// output_width.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    double values[] = { 1.23, 35.36, 653.7, 4358.24 };
    for( int i = 0; i < 4; i++ )
    {
        cout.width(10);
        cout << values[i] << '\n';
    }
}
```

```
    1.23
   35.36
  653.7
4358.24
```

Leading blanks are added to any value fewer than 10 characters wide.

To pad a field, use the `fill` member function, which sets the value of the padding character for fields that have a specified width. The default is a blank. To pad the column of numbers with asterisks, modify the previous **for** loop as follows:

```
for (int i = 0; i < 4; i++)
{
    cout.width(10);
    cout.fill('*');
    cout << values[i] << endl;
}
```

The `endl` manipulator replaces the newline character (`'\n'`). The output looks like this:

```
*****1.23
*****35.36
*****653.7
***4358.24
```

To specify widths for data elements in the same line, use the `setw` manipulator:

```
// setw.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>
using namespace std;

int main( )
{
    double values[] = { 1.23, 35.36, 653.7, 4358.24 };
    char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
    for( int i = 0; i < 4; i++ )
        cout << setw( 7 ) << names[i]
            << setw( 10 ) << values[i] << endl;
}
```

The `width` member function is declared in `<iostream>`. If you use `setw` or any other manipulator with arguments, you must include `<iomanip>`. In the output, strings are printed in a field of width 6 and integers in a field of width 10:

```

Zoot      1.23
Jimmy     35.36
  Al      653.7
Stan    4358.24
```

Neither `setw` nor `width` truncates values. If formatted output exceeds the width, the entire value prints, subject to the stream's precision setting. Both `setw` and `width` affect the following field only. Field width reverts to its default behavior (the necessary width) after one field has been printed. However, the other stream format options remain in effect until changed.

Alignment

Output streams default to right-aligned text. To left-align the names in the previous example and right-align the numbers, replace the **for** loop as follows:

```
for (int i = 0; i < 4; i++)
    cout << setiosflags(ios::left)
        << setw(6) << names[i]
        << resetiosflags(ios::left)
        << setw(10) << values[i] << endl;
```

The output looks like this:

```
Zoot      1.23
Jimmy     35.36
Al        653.7
Stan     4358.24
```

The left-align flag is set by using the `setiosflags` manipulator with the `left` enumerator. This enumerator is defined in the `ios` class, so its reference must include the `ios::` prefix. The `resetiosflags` manipulator turns off the left-align flag. Unlike `width` and `setw`, the effect of `setiosflags` and `resetiosflags` is permanent.

Precision

The default value for floating-point precision is six. For example, the number 3466.9768 prints as 3466.98. To change the way this value prints, use the `setprecision` manipulator. The manipulator has two flags: `fixed` and `scientific`. If `fixed` is set, the number prints as 3466.976800. If `scientific` is set, it prints as 3.4669773+003.

To display the floating-point numbers shown in [Alignment](#) with one significant digit, replace the **for** loop as follows:

```
for (int i = 0; i <4; i++)
    cout << setiosflags(ios::left)
          << setw(6)
          << names[i]
          << resetiosflags(ios::left)
          << setw(10)
          << setprecision(1)
          << values[i]
          << endl;
```

The program prints this list:

```
Zoot      1
Jimmy     4e+01
Al        7e+02
Stan     4e+03
```

To eliminate scientific notation, insert this statement before the **for** loop:

```
cout << setiosflags(ios::fixed);
```

With fixed notation, the program prints with one digit after the decimal point.

```
Zoot      1.2
Jimmy     35.4
Al        653.7
Stan     4358.2
```

If you change the `ios::fixed` flag to `ios::scientific`, the program prints this:

```
Zoot      1.2e+00
Jimmy     3.5e+01
Al        6.5e+02
Stan     4.4e+03
```

Again, the program prints one digit after the decimal point. If either `ios::fixed` or `ios::scientific` is set, the precision value determines the number of digits after the decimal point. If neither flag is set, the precision value determines the total number of significant digits. The `resetiosflags` manipulator clears these flags.

Radix

The `dec`, `oct`, and `hex` manipulators set the default radix for input and output. For example, if you insert the `hex` manipulator into the output stream, the object correctly translates the internal data representation of integers into a hexadecimal output format. The numbers are displayed with digits a through f in lower case if the [uppercase](#) flag is clear (the default); otherwise, they are displayed in upper case. The default radix is `dec` (decimal).

Quoted strings (C++14)

When you insert a string into a stream, you can easily retrieve the same string back by calling the `stringstream::str()` member function. However, if you want to use the extraction operator to insert the stream into a new string at a later point, you may get an unexpected result because the `>>` operator by default will stop when it encounters the first whitespace character.

```
std::stringstream ss;
std::string inserted = "This is a sentence.";
std::string extracted;

ss << inserted;
ss >> extracted;

std::cout << inserted;    // This is a sentence.
std::cout << extracted;   // This
```

This behavior can be overcome manually, but to make string round-tripping more convenient, C++14 adds the `std::quoted` stream manipulator in `<iomanip>`. Upon insertion, `quoted()` surrounds the string with a delimiter (double quote `' '` by default) and upon extraction manipulates the stream to extract all characters until the final delimiter is encountered. Any embedded quotes are escaped with an escape character (`'\\'` by default).

The delimiters are present only in the stream object; they are not present in the extracted string but they are present in the string returned by [basic_stringstream::str](#).

The whitespace behavior of the insertion and extraction operations is independent of how a string is represented in code, so the `quoted` operator is useful regardless of whether the input string is a raw string literal or a regular string. The input string, whatever its format, can have embedded quotes, line breaks, tabs, and so on and all these will be preserved by the `quoted()` manipulator.

For more information and full code examples, see [quoted](#).

See also

[Output Streams](#)

Output File Stream Member Functions

10/31/2018 • 3 minutes to read • [Edit Online](#)

Output stream member functions have three types: those that are equivalent to manipulators, those that perform unformatted write operations, and those that otherwise modify the stream state and have no equivalent manipulator or insertion operator. For sequential, formatted output, you might use only insertion operators and manipulators. For random-access binary disk output, you use other member functions, with or without insertion operators.

The open Function for Output Streams

To use an output file stream ([ofstream](#)), you must associate that stream with a specific disk file in the constructor or the `open` function. If you use the `open` function, you can reuse the same stream object with a series of files. In either case, the arguments describing the file are the same.

When you open the file associated with an output stream, you generally specify an `open_mode` flag. You can combine these flags, which are defined as enumerators in the `ios` class, with the bitwise OR (`|`) operator. See [ios_base::openmode](#) for a list of the enumerators.

Three common output stream situations involve mode options:

- Creating a file. If the file already exists, the old version is deleted.

```
ostream ofile("FILENAME");  
// Default is ios::out  
  
ofstream ofile("FILENAME", ios::out);  
// Equivalent to above
```

- Appending records to an existing file or creating one if it does not exist.

```
ofstream ofile("FILENAME", ios::app);
```

- Opening two files, one at a time, on the same stream.

```
ofstream ofile();  
ofile.open("FILE1", ios::in);  
// Do some output  
ofile.close();    // FILE1 closed  
ofile.open("FILE2", ios::in);  
// Do some more output  
ofile.close();    // FILE2 closed  
// When ofile goes out of scope it is destroyed.
```

The put

The **put** function writes one character to the output stream. The following two statements are the same by default, but the second is affected by the stream's format arguments:


```
cout.put('A');

// Exactly one character written
cout <<'A'; // Format arguments 'width' and 'fill' apply
```

The write

The `write` function writes a block of memory to an output file stream. The length argument specifies the number of bytes written. This example creates an output file stream and writes the binary value of the `Date` structure to it:

```
// write_function.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;

struct Date
{
    int mo, da, yr;
};

int main( )
{
    Date dt = { 6, 10, 92 };
    ofstream tfile( "date.dat" , ios::binary );
    tfile.write( (char *) &dt, sizeof dt );
}
```

The `write` function does not stop when it reaches a null character, so the complete class structure is written. The function takes two arguments: a **char** pointer and a count of characters to write. Note the required cast to **char*** before the address of the structure object.

The seekp and tellp Functions

An output file stream keeps an internal pointer that points to the position where data is to be written next. The `seekp` member function sets this pointer and thus provides random-access disk file output. The `tellp` member function returns the file position. For examples that use the input stream equivalents to `seekp` and `tellp`, see [The seekg and tellg Functions](#).

The close Function for Output Streams

The `close` member function closes the disk file associated with an output file stream. The file must be closed to complete all disk output. If necessary, the `ofstream` destructor closes the file for you, but you can use the `close` function if you need to open another file for the same stream object.

The output stream destructor automatically closes a stream's file only if the constructor or the `open` member function opened the file. If you pass the constructor a file descriptor for an already-open file or use the `attach` member function, you must close the file explicitly.

Error Processing Functions

Use these member functions to test for errors while writing to a stream:

FUNCTION	RETURN VALUE
<code>bad</code>	Returns true if there is an unrecoverable error.

FUNCTION	RETURN VALUE
<code>fail</code>	Returns true if there is an unrecoverable error or an "expected" condition, such as a conversion error, or if the file is not found. Processing can often resume after a call to <code>clear</code> with a zero argument.
<code>good</code>	Returns true if there is no error condition (unrecoverable or otherwise) and the end-of-file flag is not set.
<code>eof</code>	Returns true on the end-of-file condition.
<code>clear</code>	Sets the internal error state. If called with the default arguments, it clears all error bits.
<code>[rdstate](basic-ios-class.md#rdstate)</code>	Returns the current error state.

The **!** operator is overloaded to perform the same function as the `fail` function. Thus the expression:

```
if(!cout)...
```

is equivalent to:

```
if(cout.fail())...
```

The **void*()** operator is overloaded to be the opposite of the **!** operator; thus the expression:

```
if(cout)...
```

is equal to:

```
if(!cout.fail())...
```

The **void*()** operator is not equivalent to `good` because it does not test for the end of file.

See also

[Output Streams](#)

Effects of Buffering

10/31/2018 • 2 minutes to read • [Edit Online](#)

The following example shows the effects of buffering. You might expect the program to print `please wait`, wait 5 seconds, and then proceed. It will not necessarily work this way, however, because the output is buffered.

```
// effects_buffering.cpp
// compile with: /EHsc
#include <iostream>
#include <time.h>
using namespace std;

int main( )
{
    time_t tm = time( NULL ) + 5;
    cout << "Please wait...";
    while ( time( NULL ) < tm )
        ;
    cout << "\nAll done" << endl;
}
```

To make the program work logically, the `cout` object must empty itself when the message is to appear. To flush an `ostream` object, send it the `flush` manipulator:

```
cout <<"Please wait..." <<flush;
```

This step flushes the buffer, ensuring the message prints before the wait. You can also use the `endl` manipulator, which flushes the buffer and outputs a carriage return-linefeed, or you can use the `cin` object. This object (with the `cerr` or `clog` objects) is usually tied to the `cout` object. Thus, any use of `cin` (or of the `cerr` or `clog` objects) flushes the `cout` object.

See also

[Output Streams](#)

Binary Output Files

10/31/2018 • 2 minutes to read • [Edit Online](#)

Streams were originally designed for text, so the default output mode is text. In text mode, the newline character (hexadecimal 10) expands to a carriage return-linefeed (16-bit only). The expansion can cause problems, as shown here:

```
// binary_output_files.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };
int main( )
{
    ofstream os( "test.dat" );
    os.write( (char *) iarray, sizeof( iarray ) );
}
```

You might expect this program to output the byte sequence { 99, 0, 10, 0 }; instead, it outputs { 99, 0, 13, 10, 0 }, which causes problems for a program expecting binary input. If you need true binary output, in which characters are written untranslated, you could specify binary output by using the [ofstream](#) constructor openmode argument:

```
// binary_output_files2.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };

int main()
{
    ofstream ofs ( "test.dat", ios_base::binary );

    // Exactly 8 bytes written
    ofs.write( (char*)&iarray[0], sizeof(int)*2 );
}
```

See also

[Output Streams](#)

Overloading the << Operator for Your Own Classes

10/31/2018 • 2 minutes to read • [Edit Online](#)

Output streams use the insertion (`<<`) operator for standard types. You can also overload the `<<` operator for your own classes.

Example

The `write` function example showed the use of a `Date` structure. A date is an ideal candidate for a C++ class in which the data members (month, day, and year) are hidden from view. An output stream is the logical destination for displaying such a structure. This code displays a date using the `cout` object:

```
Date dt(1, 2, 92);

cout << dt;
```

To get `cout` to accept a `Date` object after the insertion operator, overload the insertion operator to recognize an `ostream` object on the left and a `Date` on the right. The overloaded `<<` operator function must then be declared as a friend of class `Date` so it can access the private data within a `Date` object.

```
// overload_date.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }
    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt)
{
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

int main()
{
    Date dt(5, 6, 92);
    cout << dt;
}
```

```
5/6/92
```

Remarks

The overloaded operator returns a reference to the original `ostream` object, which means you can combine

insertions:

```
cout <<"The date is" <<dt <<flush;
```

See also

[Output Streams](#)

Writing Your Own Manipulators Without Arguments

10/31/2018 • 2 minutes to read • [Edit Online](#)

Writing manipulators that do not use arguments requires neither class derivation nor use of complex macros. Suppose your printer requires the pair <ESC>[to enter bold mode. You can insert this pair directly into the stream:

```
cout << "regular " << '\033' << '[' << "boldface" << endl;
```

Or you can define the `bold` manipulator, which inserts the characters:

```
ostream& bold(ostream& os) {  
    return os << '\033' << '[';  
}  
cout << "regular " << bold << "boldface" << endl;
```

The globally defined `bold` function takes an `ostream` reference argument and returns the `ostream` reference. It is not a member function or a friend because it does not need access to any private class elements. The `bold` function connects to the stream because the stream's `<<` operator is overloaded to accept that type of function, using a declaration that looks something like this:

```
_Myt& operator<<(ios_base& (__cdecl *_Pfn)(ios_base&))  
{  
    // call ios_base manipulator  
    (*_Pfn)(*(ios_base *)this);  
  
    return (*this);  
}
```

You can use this feature to extend other overloaded operators. In this case, it is incidental that `bold` inserts characters into the stream. The function is called when it is inserted into the stream, not necessarily when the adjacent characters are printed. Thus, printing could be delayed because of the stream's buffering.

See also

[Output Streams](#)

Input Streams

10/31/2018 • 2 minutes to read • [Edit Online](#)

An input stream object is a source of bytes. The three most important input stream classes are `istream`, `ifstream`, and `istringstream`.

The `istream` class is best used for sequential text-mode input. You can configure objects of class `istream` for buffered or unbuffered operation. All functionality of the base class, `ios`, is included in `istream`. You will rarely construct objects from class `istream`. Instead, you will generally use the predefined `cin` object, which is actually an object of class `ostream`. In some cases, you can assign `cin` to other stream objects after program startup.

The `ifstream` class supports disk file input. If you need an input-only disk file, construct an object of class `ifstream`. You can specify binary or text-mode data. If you specify a filename in the constructor, the file is automatically opened when the object is constructed. Otherwise, you can use the `open` function after invoking the default constructor. Many formatting options and member functions apply to `ifstream` objects. All functionality of the base classes `ios` and `istream` is included in `ifstream`.

Like the library function `sscanf_s`, the `istringstream` class supports input from in-memory strings. To extract data from a character array that has a null terminator, allocate and initialize the string, then construct an object of class `istringstream`.

In This Section

[Constructing Input Stream Objects](#)

[Using Extraction Operators](#)

[Testing for Extraction Errors](#)

[Input Stream Manipulators](#)

[Input Stream Member Functions](#)

[Overloading the >> Operator for Your Own Classes](#)

See also

[iostream Programming](#)

Constructing Input Stream Objects

10/31/2018 • 2 minutes to read • [Edit Online](#)

If you use only the `cin` object, you do not need to construct an input stream. You must construct an input stream if you use:

- [Input File Stream Constructors](#)
- [Input String Stream Constructors](#)

Input File Stream Constructors

There are two ways to create an input file stream:

- Use the **void** argument constructor, then call the `open` member function:

```
ifstream myFile; // On the stack
myFile.open("filename");

ifstream* pmyFile = new ifstream; // On the heap
pmyFile->open("filename");
```

- Specify a filename and mode flags in the constructor invocation, thereby opening the file during the construction process:

```
ifstream myFile("filename");
```

Input String Stream Constructors

Input string stream constructors require the address of preallocated, preinitialized storage:

```
string s("123.45");

double amt;
istringstream myString(s);

//istringstream myString("123.45") also works
myString >> amt; // amt contains 123.45
```

See also

[Input Streams](#)

Using Extraction Operators

10/31/2018 • 2 minutes to read • [Edit Online](#)

The extraction operator (`>>`), which is preprogrammed for all standard C++ data types, is the easiest way to get bytes from an input stream object.

Formatted text input extraction operators depend on white space to separate incoming data values. This is inconvenient when a text field contains multiple words or when commas separate numbers. In such a case, one alternative is to use the unformatted input member function `istream::getline` to read a block of text with white space included, then parse the block with special functions. Another method is to derive an input stream class with a member function such as `GetNextToken`, which can call `istream` members to extract and format character data.

See also

[Input Streams](#)

Testing for Extraction Errors

10/31/2018 • 2 minutes to read • [Edit Online](#)

Output error processing functions, discussed in [Error Processing Functions](#), apply to input streams. Testing for errors during extraction is important. Consider this statement:

```
cin >> n;
```

If `n` is a signed integer, a value greater than 32,767 (the maximum allowed value, or `MAX_INT`) sets the stream's `fail` bit, and the `cin` object becomes unusable. All subsequent extractions result in an immediate return with no value stored.

See also

[Input Streams](#)

Input Stream Manipulators

10/31/2018 • 2 minutes to read • [Edit Online](#)

Many manipulators, such as [setprecision](#), are defined for the `ios` class and thus apply to input streams. Few manipulators, however, actually affect input stream objects. Of those that do, the most important are the radix manipulators, `dec`, `oct`, and `hex`, which determine the conversion base used with numbers from the input stream.

On extraction, the `hex` manipulator enables processing of various input formats. For example, `c`, `C`, `0xc`, `0xC`, `0Xc`, and `0XC` are all interpreted as the decimal integer 12. Any character other than 0 through 9, A through F, a through f, `x`, and `X` terminates the numeric conversion. Thus the sequence `"124n5"` is converted to the number 124 with the [basic_ios::fail](#) bit set.

See also

[Input Streams](#)

Input Stream Member Functions

10/31/2018 • 4 minutes to read • [Edit Online](#)

Input stream member functions are used for disk input. The member functions include:

- [The open Function for Input Streams](#)
- [The get](#)
- [The getline](#)
- [The read](#)
- [The seekg and tellg Functions](#)
- [The close Function for Input Streams](#)

The open Function for Input Streams

If you are using an input file stream (ifstream), you must associate that stream with a specific disk file. You can do this in the constructor, or you can use the `open` function. In either case, the arguments are the same.

You generally specify an `ios_base::openmode` flag when you open the file associated with an input stream (the default mode is `ios::in`). For a list of the `open_mode` flags, see [The open](#). The flags can be combined with the bitwise OR (`|`) operator.

To read a file, first use the `fail` member function to determine whether it exists:

```
istream ifile("FILENAME");

if (ifile.fail())
    // The file does not exist ...
```

The get

The unformatted `get` member function works like the `>>` operator with two exceptions. First, the `get` function includes white-space characters, whereas the extractor excludes white space when the `skipws` flag is set (the default). Second, the `get` function is less likely to cause a tied output stream (`cout`, for example) to be flushed.

A variation of the `get` function specifies a buffer address and the maximum number of characters to read. This is useful for limiting the number of characters sent to a specific variable, as this example shows:

```
// io_get_function.cpp
// compile with: /EHsc
// Type up to 24 characters and a terminating character.
// Any remaining characters can be extracted later.
#include <iostream>
using namespace std;

int main()
{
    char line[25];
    cout << " Type a line terminated by carriage return\n>";
    cin.get( line, 25 );
    cout << line << endl;
}
```

Input

1234

Sample Output

1234

The getline

The `getline` member function is similar to the `get` function. Both functions allow a third argument that specifies the terminating character for input. The default value is the newline character. Both functions reserve one character for the required terminating character. However, `get` leaves the terminating character in the stream and `getline` removes the terminating character.

The following example specifies a terminating character for the input stream:

```
// getline_func.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
    char line[100];
    cout << " Type a line terminated by 't'" << endl;
    cin.getline( line, 100, 't' );
    cout << line;
}
```

Input

test

The read

The `read` member function reads bytes from a file to a specified area of memory. The length argument determines the number of bytes read. If you do not include that argument, reading stops when the physical end of file is reached or, in the case of a text-mode file, when an embedded `EOF` character is read.

This example reads a binary record from a payroll file into a structure:

```

#include <fstream>
#include <iostream>
using namespace std;

int main()
{
    struct
    {
        double salary;
        char name[23];
    } employee;

    ifstream is( "payroll" );
    if( is ) { // ios::operator void*()
        is.read( (char *) &employee, sizeof( employee ) );
        cout << employee.name << ' ' << employee.salary << endl;
    }
    else {
        cout << "ERROR: Cannot open file 'payroll'." << endl;
    }
}

```

The program assumes that the data records are formatted exactly as specified by the structure with no terminating carriage-return or linefeed characters.

The seekg and tellg Functions

Input file streams keep an internal pointer to the position in the file where data is to be read next. You set this pointer with the `seekg` function, as shown here:

```

#include <iostream>
#include <fstream>
using namespace std;

int main( )
{
    char ch;

    ifstream tfile( "payroll" );
    if( tfile ) {
        tfile.seekg( 8 ); // Seek 8 bytes in (past salary)
        while ( tfile.good() ) { // EOF or failure stops the reading
            tfile.get( ch );
            if( !ch ) break; // quit on null
            cout << ch;
        }
    }
    else {
        cout << "ERROR: Cannot open file 'payroll'." << endl;
    }
}

```

To use `seekg` to implement record-oriented data management systems, multiply the fixed-length record size by the record number to obtain the byte position relative to the end of the file, and then use the `get` object to read the record.

The `tellg` member function returns the current file position for reading. This value is of type `streampos`, a `typedef` defined in `<iostream>`. The following example reads a file and displays messages showing the positions of spaces.

```

#include <fstream>
#include <iostream>
using namespace std;

int main( )
{
    char ch;
    ifstream tfile( "payroll" );
    if( tfile ) {
        while ( tfile.good( ) ) {
            streampos here = tfile.tellg();
            tfile.get( ch );
            if ( ch == ' ' )
                cout << "\nPosition " << here << " is a space";
        }
    }
    else {
        cout << "ERROR: Cannot open file 'payroll'." << endl;
    }
}

```

The close Function for Input Streams

The `close` member function closes the disk file associated with an input file stream and frees the operating system file handle. The `ifstream` destructor closes the file for you, but you can use the `close` function if you need to open another file for the same stream object.

See also

[Input Streams](#)

Overloading the >> Operator for Your Own Classes

10/31/2018 • 2 minutes to read • [Edit Online](#)

Input streams use the extraction (`>>`) operator for the standard types. You can write similar extraction operators for your own types; your success depends on using white space precisely.

Here is an example of an extraction operator for the `Date` class presented earlier:

```
istream& operator>> (istream& is, Date& dt)
{
    is>> dt.mo>> dt.da>> dt.yr;
    return is;
}
```

See also

[Input Streams](#)

Input/Output Streams

10/31/2018 • 2 minutes to read • [Edit Online](#)

`basic_istream`, which is defined in the header file `<istream>`, is the class template for objects that handle both input and output character-based I/O streams.

There are two typedefs that define character-specific specializations of `basic_istream` and can help make code easier to read: `istream` (not to be confused with the header file `<istream>`) is an I/O stream that is based on `basic_istream<char>`; `wistream` is an I/O stream that is based on `basic_istream<wchar_t>`.

For more information, see [basic_istream Class](#), [istream](#), and [wistream](#).

Deriving from `basic_istream` is the class template `basic_fstream`, which is used to stream character data to and from files.

There also are typedefs that provide character-specific specializations of `basic_fstream`. They are `fstream`, which is a file I/O stream that is based on **char**, and `wfstream`, which is a file I/O stream that is based on **wchar_t**. For more information, see [basic_fstream Class](#), [fstream](#), and [wfstream](#). Using these typedefs requires the inclusion of the header file `<fstream>`.

NOTE

When a `basic_fstream` object is used to perform file I/O, although the underlying buffer contains separately designated positions for reading and writing, the current input and current output positions are tied together, and therefore, reading some data moves the output position.

The class template `basic_stringstream` and its common specialization, `stringstream`, are often used to work with I/O stream objects to insert and extract character data. For more information, see [basic_stringstream Class](#).

See also

[stringstream](#)

[basic_stringstream Class](#)

[<sstream>](#)

[istream Programming](#)

[C++ Standard Library](#)

iostreams Conventions

10/31/2018 • 2 minutes to read • [Edit Online](#)

The iostreams headers support conversions between text and encoded forms, and input and output to external files:

<fstream>	<iomanip>
<ios>	<iosfwd>
<iostream>	<istream>
<ostream>	<sstream>
<streambuf>	<strstream>

The simplest use of iostreams requires only that you include the header [<iostream>](#). You can then extract values from [cin](#) or [wcin](#) to read the standard input. The rules for doing so are outlined in the description of the class [basic_istream Class](#). You can also insert values to [cout](#) or [wcout](#) to write the standard output. The rules for doing so are outlined in the description of the class [basic_ostream Class](#). Format control common to both extractors and insertors is managed by the class [basic_ios Class](#). Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators.

You can perform the same iostreams operations on files that you open by name, using the classes declared in [<fstream>](#). To convert between iostreams and objects of class [basic_string Class](#), use the classes declared in [<sstream>](#). To do the same with C strings, use the classes declared in [<strstream>](#).

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

See also

[C++ Standard Library Overview](#)

[iostream Programming](#)

[Thread Safety in the C++ Standard Library](#)

Custom Manipulators with Arguments

10/31/2018 • 2 minutes to read • [Edit Online](#)

This section describes how to create output stream manipulators with one or more arguments, and how to use manipulators for nonoutput streams.

In This Section

[Output Stream Manipulators with One Argument \(**int** or **long**\)](#)

[Other One-Argument Output Stream Manipulators](#)

See also

[iostream Programming](#)

Output Stream Manipulators with One Argument (int or long)

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `iostream` class library provides a set of macros for creating parameterized manipulators. Manipulators with a single **int** or **long** argument are a special case. To create an output stream manipulator that accepts a single **int** or **long** argument (like `setw`), you must use the `_Smanip` macro, which is defined in `<iomanip>`. This example defines a `fillblank` manipulator that inserts a specified number of blanks into the stream:

Example

```
// output_stream_manip.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
using namespace std;

void fb( ios_base& os, int l )
{
    ostream *pos = dynamic_cast<ostream*>(&os);
    if (pos)
    {
        for( int i=0; i < l; i++ )
            (*pos) << ' ';
    };
}

_Smanip<int>
__cdecl fillblank(int no)
{
    return (_Smanip<int>(&fb, no));
}

int main( )
{
    cout << "10 blanks follow" << fillblank( 10 ) << ".\n";
}
```

See also

[Custom Manipulators with Arguments](#)

Other One-Argument Output Stream Manipulators

10/31/2018 • 2 minutes to read • [Edit Online](#)

The following example uses a class `money`, which is a **long** type. The `setpic` manipulator attaches a formatting "picture" string to the class that can be used by the overloaded stream insertion operator of the class `money`. The picture string is stored as a static variable in the `money` class rather than as data member of a stream class, so you do not have to derive a new output stream class.

Example

```

// one_arg_output.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

typedef char* charp;

class money
{
private:
    long value;
    static char *szCurrentPic;
public:
    money( long val ) { value = val; }
    friend ostream& operator << ( ostream& os, money m ) {
        // A more complete function would merge the picture
        // with the value rather than simply appending it
        os << m.value << '[' << money::szCurrentPic << ']' ;
        return os;
    }
    static void setpic( char* szPic ) {
        money::szCurrentPic = new char[strlen( szPic ) + 1];
        strcpy_s( money::szCurrentPic, strlen( szPic ) + 1, szPic );
    }
};

char *money::szCurrentPic; // Static pointer to picture

void fb( ios_base& os, char * somename )
{
    money::setpic(somename);
    /*
    ostream *pos = dynamic_cast<ostream*>(&os);
    if (pos)
    {
        for( int i=0; i < 1; i++ )
            (*pos) << ' ';
    }
    */
}

_Smanip<charp>
__cdecl setpic(char * somename)
{
    return (_Smanip<charp>(&fb, somename));
}

int main( )
{
    money amt = (long)35235.22;
    cout << setiosflags( ios::fixed );
    cout << setpic( "###,###,###.##" ) << "amount = " << amt << endl;
}

```

See also

[Custom Manipulators with Arguments](#)

Regular Expressions (C++)

5/7/2019 • 24 minutes to read • [Edit Online](#)

The C++ standard library supports multiple regular expression grammars. This topic discusses the grammar variations available when using regular expressions.

Regular Expression Grammar

The regular expression grammar to use is specified by the use of one of the

`std::regex_constants::syntax_option_type` enumeration values. These regular expression grammars are defined in `std::regex_constants`:

- `ECMAScript`: This is closest to the grammar used by JavaScript and the .NET languages.
- `basic`: The POSIX basic regular expressions or BRE.
- `extended`: The POSIX extended regular expressions or ERE.
- `awk`: This is `extended`, but it has additional escapes for non-printing characters.
- `grep`: This is `basic`, but it also allows newline (`'\n'`) characters to separate alternations.
- `egrep`: This is `extended`, but it also allows newline characters to separate alternations.

By default, if no grammar is specified, `ECMAScript` is assumed. Only one grammar may be specified.

In addition to the grammar, several flags can be applied:

- `icase`: Ignore case when matching.
- `nosubs`: Ignore marked matches (that is, expressions in parentheses); no substitutions are stored.
- `optimize`: Make matching faster, at the possible expense of greater construction time.
- `collate`: Use locale-sensitive collation sequences (for example, ranges of the form `"[a-z]"`).

Zero or more flags may be combined with the grammar to specify the regular expression engine behavior. If only flags are specified, `ECMAScript` is assumed as the grammar.

Element

An element can be one of the following things:

- An *ordinary character* that matches the same character in the target sequence.
- A *wildcard character* `'.'` that matches any character in the target sequence except a newline.
- A *bracket expression* of the form `"[expr]"`, which matches a character or a collation element in the target sequence that is also in the set defined by the expression `expr`, or of the form `"[^ expr]"`, which matches a character or a collation element in the target sequence that is not in the set defined by the expression `expr`.

The expression `expr` can contain any combination of the following things:

- An individual character. Adds that character to the set defined by `expr`.
- A *character range* of the form `"ch1 - ch2"`. Adds the characters that are represented by values in the closed range `[ch1 , ch2]` to the set defined by `expr`.
- A *character class* of the form `"[: name :]"`. Adds the characters in the named class to the set defined by `expr`.

- An *equivalence class* of the form "[= `elt` =]". Adds the collating elements that are equivalent to `elt` to the set defined by `expr`.
- A *collating symbol* of the form "[. `elt` .]". Adds the collation element `elt` to the set defined by `expr`.
- An *anchor*. Anchor '^' matches the beginning of the target sequence; anchor '\$' matches the end of the target sequence.

A *capture group* of the form "(*subexpression*)", or "\ (*subexpression* \)" in `basic` and `grep`, which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.

- An *identity escape* of the form "\ `k`", which matches the character `k` in the target sequence.

Examples:

- "a" matches the target sequence "a" but does not match the target sequences "B", "b", or "c".
- "." matches all the target sequences "a", "B", "b", and "c".
- "[b-z]" matches the target sequences "b" and "c" but does not match the target sequences "a" or "B".
- "[:lower:]" matches the target sequences "a", "b", and "c" but does not match the target sequence "B".
- "(a)" matches the target sequence "a" and associates capture group 1 with the subsequence "a", but does not match the target sequences "B", "b", or "c".

In `ECMAScript`, `basic`, and `grep`, an element can also be a *back reference* of the form "\ `dd`", where `dd` represents a decimal value N that matches a sequence of characters in the target sequence that is the same as the sequence of characters that is matched by the Nth *capture group*. For example, "(a)\1" matches the target sequence "aa" because the first (and only) capture group matches the initial sequence "a" and then the \1 matches the final sequence "a".

In `ECMAScript`, an element can also be one of the following things:

- A *non-capture group* of the form "(?: *subexpression*)". Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.
- A limited *file format escape* of the form "\f", "\n", "\r", "\t", or "\v". These match a form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- A *positive assert* of the form "(= *subexpression*)". Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters, but does not change the match position in the target sequence.
- A *negative assert* of the form "(! *subexpression*)". Matches any sequence of characters in the target sequence that does not match the pattern between the delimiters, and does not change the match position in the target sequence.
- A *hexadecimal escape sequence* of the form "\x `hh`". Matches a character in the target sequence that is represented by the two hexadecimal digits `hh`.
- A *unicode escape sequence* of the form "\u `hhhh`". Matches a character in the target sequence that is represented by the four hexadecimal digits `hhhh`.
- A *control escape sequence* of the form "\c `k`". Matches the control character that is named by the character `k`.
- A *word boundary assert* of the form "\b". Matches when the current position in the target sequence is immediately after a *word boundary*.

- A *negative word boundary assert* of the form `"\B"`. Matches when the current position in the target sequence is not immediately after a *word boundary*.
- A *dsw character escape* of the form `"\d"`, `"\D"`, `"\s"`, `"\S"`, `"\w"`, `"\W"`. Provides a short name for a character class.

Examples:

- `"(?:a)"` matches the target sequence `"a"`, but `"(?:a)\1"` is invalid because there is no capture group 1.
- `"(=a)a"` matches the target sequence `"a"`. The positive assert matches the initial sequence `"a"` in the target sequence and the final `"a"` in the regular expression matches the initial sequence `"a"` in the target sequence.
- `"(!a)a"` does not match the target sequence `"a"`.
- `"a\b."` matches the target sequence `"a~"`, but does not match the target sequence `"ab"`.
- `"a\B."` matches the target sequence `"ab"`, but does not match the target sequence `"a~"`.

In `awk`, an element can also be one of the following things:

- A *file format escape* of the form `"\\"`, `"\a"`, `"\b"`, `"\f"`, `"\n"`, `"\r"`, `"\t"`, or `"\v"`. These match a backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- An *octal escape sequence* of the form `"\ooo"`. Matches a character in the target sequence whose representation is the value represented by the one, two, or three octal digits `ooo`.

Repetition

Any element other than a *positive assert*, a *negative assert*, or an *anchor* can be followed by a repetition count. The most general kind of repetition count takes the form `"{min, max}"`, or `"{\min, max}"` in `basic` and `grep`. An element that is followed by this form of repetition count matches at least `min` successive occurrences and no more than `max` successive occurrences of a sequence that matches the element. For example, `"a{2,3}"` matches the target sequence `"aa"` and the target sequence `"aaa"`, but not the target sequence `"a"` or the target sequence `"aaaa"`.

A repetition count can also take one of the following forms:

- `"{min}"`, or `"{\min}"` in `basic` and `grep`. Equivalent to `"{min, min}"`.
- `"{min,}"`, or `"{\min,}"` in `basic` and `grep`. Equivalent to `"{min, unbounded}"`.
- `"*"`. Equivalent to `"{0, unbounded}"`.

Examples:

- `"a{2}"` matches the target sequence `"aa"` but not the target sequence `"a"` or the target sequence `"aaa"`.
- `"a{2,}"` matches the target sequence `"aa"`, the target sequence `"aaa"`, and so on, but does not match the target sequence `"a"`.
- `"a*"` matches the target sequence `""`, the target sequence `"a"`, the target sequence `"aa"`, and so on.

For all grammars except `basic` and `grep`, a repetition count can also take one of the following forms:

- `"?"`. Equivalent to `"{0,1}"`.
- `"+"`. Equivalent to `"{1, unbounded}"`.

Examples:

- `"a?"` matches the target sequence `""` and the target sequence `"a"`, but not the target sequence `"aa"`.

- "a+" matches the target sequence "a", the target sequence "aa", and so on, but not the target sequence "".

In `ECMAScript`, all the forms of repetition count can be followed by the character '?', which designates a *non-greedy repetition*.

Concatenation

Regular expression elements, with or without *repetition counts*, can be concatenated to form longer regular expressions. The resulting expression matches a target sequence that is a concatenation of the sequences that are matched by the individual elements. For example, "a{2,3}b" matches the target sequence "aab" and the target sequence "aaab", but does not match the target sequence "ab" or the target sequence "aaaab".

Alternation

In all regular expression grammars except `basic` and `grep`, a concatenated regular expression can be followed by the character '|' and another concatenated regular expression. Any number of concatenated regular expressions can be combined in this manner. The resulting expression matches any target sequence that matches one or more of the concatenated regular expressions.

When more than one of the concatenated regular expressions matches the target sequence, `ECMAScript` chooses the first of the concatenated regular expressions that matches the sequence as the match (*first match*); the other regular expression grammars choose the one that achieves the *longest match*. For example, "ab|cd" matches the target sequence "ab" and the target sequence "cd", but does not match the target sequence "abd" or the target sequence "acd".

In `grep` and `egrep`, a newline character ('\n') can be used to separate alternations.

Subexpression

In `basic` and `grep`, a subexpression is a concatenation. In the other regular expression grammars, a subexpression is an alternation.

Grammar Summary

The following table summarizes the features that are available in the various regular expression grammars:

ELEMENT	BASIC	EXTENDED	ECMASCRIPT	GREP	EGREP	AWK
alternation using ' '		+	+		+	+
alternation using '\n'				+	+	
anchor	+	+	+	+	+	+
back reference	+		+	+		
bracket expression	+	+	+	+	+	+
capture group using "()"		+	+		+	+
capture group using "\(\\""	+			+		

ELEMENT	BASIC	EXTENDED	ECMASCRIPT	GREP	EGREP	AWK
control escape sequence			+			
dsw character escape			+			
file format escape			+			+
hexadecimal escape sequence			+			
identity escape	+	+	+	+	+	+
negative assert			+			
negative word boundary assert			+			
non-capture group			+			
non-greedy repetition			+			
octal escape sequence						+
ordinary character	+	+	+	+	+	+
positive assert			+			
repetition using "{}"		+	+		+	+
repetition using "\\{}"	+			+		
repetition using '*'	+	+	+	+	+	+
repetition using '?' and '+'		+	+		+	+
unicode escape sequence			+			

ELEMENT	BASIC	EXTENDED	ECMASCRIPT	GREP	EGREP	AWK
wildcard character	+	+	+	+	+	+
word boundary assert			+			

Semantic Details

Anchor

An anchor matches a position in the target string, not a character. A '^' matches the beginning of the target string, and a '\$' matches the end of the target string.

Back Reference

A back reference is a backslash that is followed by a decimal value N. It matches the contents of the Nth *capture group*. The value of N must not be more than the number of capture groups that precede the back reference. In `basic` and `grep`, the value of N is determined by the decimal digit that follows the backslash. In `ECMAScript`, the value of N is determined by all the decimal digits that immediately follow the backslash. Therefore, in `basic` and `grep`, the value of N is never more than 9, even if the regular expression has more than nine capture groups. In `ECMAScript`, the value of N is unbounded.

Examples:

- `"((a+)(b+))(c+)\3"` matches the target sequence `"aabbbcbbb"`. The back reference `"\3"` matches the text in the third capture group, that is, the `"(b+)"`. It does not match the target sequence `"aabbbcb"`.
- `"(a)\2"` is not valid.
- `"(b((((((((a))))))))))\10"` has different meanings in `basic` and in `ECMAScript`. In `basic` the back reference is `"\1"`. The back reference matches the contents of the first capture group (that is, the one that begins with `"(b"` and ends with the final `)"` and comes before the back reference), and the final `'0'` matches the ordinary character `'0'`. In `ECMAScript`, the back reference is `"\10"`. It matches the tenth capture group, that is, the innermost one.

Bracket Expression

A bracket expression defines a set of characters and *collating elements*. When the bracket expression begins with the character '^' the match succeeds if no elements in the set match the current character in the target sequence. Otherwise, the match succeeds if any one of the elements in the set matches the current character in the target sequence.

The set of characters can be defined by listing any combination of *individual characters*, *character ranges*, *character classes*, *equivalence classes*, and *collating symbols*.

Capture Group

A capture group marks its contents as a single unit in the regular expression grammar and labels the target text that matches its contents. The label that is associated with each capture group is a number, which is determined by counting the opening parentheses that mark capture groups up to and including the opening parenthesis that marks the current capture group. In this implementation, the maximum number of capture groups is 31.

Examples:

- `"ab+"` matches the target sequence `"abb"`, but does not match the target sequence `"abab"`.
- `"(ab)+"` does not match the target sequence `"abb"`, but matches the target sequence `"abab"`.

- `"((a+)(b+))(c+)"` matches the target sequence "aabbbc" and associates capture group 1 with the subsequence "aabbb", capture group 2 with the subsequence "aa", capture group 3 with "bbb", and capture group 4 with the subsequence "c".

Character Class

A character class in a bracket expression adds all the characters in the named class to the character set that is defined by the bracket expression. To create a character class, use `"[:"` followed by the name of the class followed by `"]"`. Internally, names of character classes are recognized by calling `id = traits.lookup_classname`. A character `ch` belongs to such a class if `traits.isctype(ch, id)` returns true. The default `regex_traits` template supports the class names in the following table.

CLASS NAME	DESCRIPTION
"alnum"	lowercase letters, uppercase letters, and digits
"alpha"	lowercase letters and uppercase letters
"blank"	space or tab
"cntrl"	the <i>file format escape</i> characters
"digit"	digits
"graph"	lowercase letters, uppercase letters, digits, and punctuation
"lower"	lowercase letters
"print"	lowercase letters, uppercase letters, digits, punctuation, and space
"punct"	punctuation
"space"	space
"upper"	uppercase characters
"xdigit"	digits, 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'
"d"	same as digit
"s"	same as space
"w"	same as alnum

Character Range

A character range in a bracket expression adds all the characters in the range to the character set that is defined by the bracket expression. To create a character range, put the character `'-'` between the first and last characters in the range. Doing this puts into the set all characters that have a numeric value that is more than or equal to the numeric value of the first character, and less than or equal to the numeric value of the last character. Notice that this set of added characters depends on the platform-specific representation of characters. If the character `'-'` occurs at the beginning or the end of a bracket expression, or as the first or last character of a character range, it represents itself.

Examples:

- "[0-7]" represents the set of characters { '0', '1', '2', '3', '4', '5', '6', '7' }. It matches the target sequences "0", "1", and so on, but not "a".
- On systems that use the ASCII character encoding, "[h-k]" represents the set of characters { 'h', 'i', 'j', 'k' }. It matches the target sequences "h", "i", and so on, but not "\x8A" or "0".
- On systems that use the EBCDIC character encoding, "[h-k]" represents the set of characters { 'h', 'i', '\x8A', '\x8B', '\x8C', '\x8D', '\x8E', '\x8F', '\x90', 'j', 'k' } ('h' is encoded as 0x88 and 'k' is encoded as 0x92). It matches the target sequences "h", "i", "\x8A", and so on, but not "0".
- "[-0-24]" represents the set of characters { '-', '0', '1', '2', '4' }.
- "[0-2-]" represents the set of characters { '0', '1', '2', '-' }.
- On systems that use the ASCII character encoding, "[+--]" represents the set of characters { '+', '-', '-' }.

However, when locale-sensitive ranges are used, the characters in a range are determined by the collation rules for the locale. Characters that collate after the first character in the definition of the range and before the last character in the definition of the range are in the set. The two end characters are also in the set.

Collating Element

A collating element is a multi-character sequence that is treated as a single character.

Collating Symbol

A collating symbol in a bracket expression adds a *collating element* to the set that is defined by the bracket expression. To create a collating symbol, use "[" followed by the collating element followed by "."].

Control Escape Sequence

A control escape sequence is a backslash followed by the letter 'c' followed by one of the letters 'a' through 'z' or 'A' through 'Z'. It matches the ASCII control character that is named by that letter. For example, "\ci" matches the target sequence "\x09", because <ctrl-i> has the value 0x09.

DSW Character Escape

A dsw character escape is a short name for a character class, as shown in the following table.

ESCAPE SEQUENCE	EQUIVALENT NAMED CLASS	DEFAULT NAMED CLASS
"\d"	"[[d:]]"	"[[digit:]]"
"\D"	"[^d:]"	"[^digit:]"
"\s"	"[[s:]]"	"[[space:]]"
"\S"	"[^s:]"	"[^space:]"
"\w"	"[[w:]]"	"[a-zA-Z0-9_]"*
"\W"	"[^w:]"	"[^a-zA-Z0-9_]"*

*ASCII character set

Equivalence Class

An equivalence class in a bracket expression adds all the characters and *collating elements* that are equivalent to the collating element in the equivalence class definition to the set that is defined by the bracket expression. To create an equivalence class, use "[=" followed by a collating element followed by "=]". Internally, two collating

elements `elt1` and `elt2` are equivalent if

```
traits.transform_primary(elt1.begin(), elt1.end()) == traits.transform_primary(elt2.begin(), elt2.end()) .
```

File Format Escape

A file format escape consists of the usual C language character escape sequences, `"\\", "\a", "\b", "\f", "\n", "\r", "\t", "\v"`. These have the usual meanings, that is, backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively. In `ECMAScript`, `"\a"` and `"\b"` are not allowed. (`"\\"` is allowed, but it is an identity escape, not a file format escape).

Hexadecimal Escape Sequence

A hexadecimal escape sequence is a backslash followed by the letter 'x' followed by two hexadecimal digits (0-9a-fA-F). It matches a character in the target sequence that has the value that is specified by the two digits. For example, `"\x41"` matches the target sequence "A" when ASCII character encoding is used.

Identity Escape

An identity escape is a backslash followed by a single character. It matches that character. It is required when the character has a special meaning; by using the identity escape, the special meaning is removed. For example:

- `"a*"` matches the target sequence "aaa", but does not match the target sequence "a*".
- `"a\""` does not match the target sequence "aaa", but matches the target sequence "a*".

The set of characters that are allowed in an identity escape depends on the regular expression grammar, as shown in the following table.

GRAMMAR	ALLOWED IDENTITY ESCAPE CHARACTERS
<code>basic</code> , <code>grep</code>	{ <code>'(,), {, }, ^, \, *, ^, \$'</code> }
<code>extended</code> , <code>egrep</code>	{ <code>'(,), {, }, ^, \, *, ^, \$, +, ?, '</code> }
<code>awk</code>	<code>extended</code> plus <code>'" , /'</code> }
<code>ECMAScript</code>	All characters except those that can be part of an identifier. Typically, this includes letters, digits, <code>'\$, _'</code> , and unicode escape sequences. For more information, see the ECMAScript Language Specification.

Individual Character

An individual character in a bracket expression adds that character to the character set that is defined by the bracket expression. Anywhere in a bracket expression except at the beginning, a `'^'` represents itself.

Examples:

- `"[abc]"` matches the target sequences "a", "b", and "c", but not the sequence "d".
- `"[^abc]"` matches the target sequence "d", but not the target sequences "a", "b", or "c".
- `"[a^bc]"` matches the target sequences "a", "b", "c", and "^", but not the target sequence "d".

In all regular expression grammars except `ECMAScript`, if a `']'` is the first character that follows the opening `'['` or is the first character that follows an initial `'^'`, it represents itself.

Examples:

- `"[a]"` is invalid because there is no `']'` to end the bracket expression.
- `"[]abc]"` matches the target sequences "a", "b", "c", and "]", but not the target sequence "d".

- "[^]abc]" matches the target sequence "d", but not the target sequences "a", "b", "c", or "]"

In `ECMAScript`, use `\]` to represent the character `]` in a bracket expression.

Examples:

- "[a]" matches the target sequence "a" because the bracket expression is empty.
- "[\]abc]" matches the target sequences "a", "b", "c", and "]" but not the target sequence "d".

Negative Assert

A negative assert matches anything but its contents. It does not consume any characters in the target sequence. For example, `!(aa)(a*)` matches the target sequence "a" and associates capture group 1 with the subsequence "a". It does not match the target sequence "aa" or the target sequence "aaa".

Negative Word Boundary Assert

A negative word boundary assert matches if the current position in the target string is not immediately after a *word boundary*.

Non-capture Group

A non-capture group marks its contents as a single unit in the regular expression grammar, but does not label the target text. For example, `(a)(?:b)*(c)` matches the target text "abbc" and associates capture group 1 with the subsequence "a" and capture group 2 with the subsequence "c".

Non-greedy Repetition

A non-greedy repetition consumes the shortest subsequence of the target sequence that matches the pattern. A greedy repetition consumes the longest. For example, `(a+)(a*b)` matches the target sequence "aaab". When a non-greedy repetition is used, it associates capture group 1 with the subsequence "a" at the beginning of the target sequence and capture group 2 with the subsequence "aab" at the end of the target sequence. When a greedy match is used, it associates capture group 1 with the subsequence "aaa" and capture group 2 with the subsequence "b".

Octal Escape Sequence

An octal escape sequence is a backslash followed by one, two, or three octal digits (0-7). It matches a character in the target sequence that has the value that is specified by those digits. If all the digits are '0', the sequence is invalid. For example, `\101` matches the target sequence "A" when ASCII character encoding is used.

Ordinary Character

An ordinary character is any valid character that does not have a special meaning in the current grammar.

In `ECMAScript`, the following characters have special meanings:

- `^ $ \ . * + ? () [] { } |`

In `basic` and `grep`, the following characters have special meanings:

- `.[\`

Also in `basic` and `grep`, the following characters have special meanings when they are used in a particular context:

- `*` has a special meaning in all cases except when it is the first character in a regular expression or the first character that follows an initial `^` in a regular expression, or when it is the first character of a capture group or the first character that follows an initial `^` in a capture group.
- `^` has a special meaning when it is the first character of a regular expression.
- `$` has a special meaning when it is the last character of a regular expression.

In `extended`, `egrep`, and `awk`, the following characters have special meanings:

- `.[\(*+?{|`

Also in `extended`, `egrep`, and `awk`, the following characters have special meanings when they are used in a particular context.

- `'` has a special meaning when it matches a preceding `'`.
- `^` has a special meaning when it is the first character of a regular expression.
- `$` has a special meaning when it is the last character of a regular expression.

An ordinary character matches the same character in the target sequence. By default, this means that the match succeeds if the two characters are represented by the same value. In a case-insensitive match, two characters `ch0` and `ch1` match if `traits.translate_nocase(ch0) == traits.translate_nocase(ch1)`. In a locale-sensitive match, two characters `ch0` and `ch1` match if `traits.translate(ch0) == traits.translate(ch1)`.

Positive Assert

A positive assert matches its contents, but does not consume any characters in the target sequence.

Examples:

- `"(=aa)(a*)"` matches the target sequence "aaaa" and associates capture group 1 with the subsequence "aaaa".
- `"(aa)(a*)"` matches the target sequence "aaaa" and associates capture group 1 with the subsequence "aa" at the beginning of the target sequence and capture group 2 with the subsequence "aa" at the end of the target sequence.
- `"(=aa)(a)|(a)"` matches the target sequence "a" and associates capture group 1 with an empty sequence (because the positive assert failed) and capture group 2 with the subsequence "a". It also matches the target sequence "aa" and associates capture group 1 with the subsequence "aa" and capture group 2 with an empty sequence.

Unicode Escape Sequence

A unicode escape sequence is a backslash followed by the letter 'u' followed by four hexadecimal digits (0-9a-fA-F). It matches a character in the target sequence that has the value that is specified by the four digits. For example, `"\u0041"` matches the target sequence "A" when ASCII character encoding is used.

Wildcard Character

A wildcard character matches any character in the target expression except a newline.

Word Boundary

A word boundary occurs in the following situations:

- The current character is at the beginning of the target sequence and is one of the word characters `A-Za-z0-9_.`
- The current character position is past the end of the target sequence and the last character in the target sequence is one of the word characters.
- The current character is one of the word characters and the preceding character is not.
- The current character is not one of the word characters and the preceding character is.

Word Boundary Assert

A word boundary assert matches when the current position in the target string is immediately after a *word boundary*.

Matching and Searching

For a regular expression to match a target sequence, the entire regular expression must match the entire target sequence. For example, the regular expression "bcd" matches the target sequence "bcd" but does not match the target sequence "abcd" nor the target sequence "bcde".

For a regular expression search to succeed, there must be a subsequence somewhere in the target sequence that matches the regular expression. The search typically finds the left-most matching subsequence.

Examples:

- A search for the regular expression "bcd" in the target sequence "bcd" succeeds and matches the entire sequence. The same search in the target sequence "abcd" also succeeds and matches the last three characters. The same search in the target sequence "bcde" also succeeds and matches the first three characters.
- A search for the regular expression "bcd" in the target sequence "bcdbcd" succeeds and matches the first three characters.

If there is more than one subsequence that matches at some location in the target sequence, there are two ways to choose the matching pattern. *First match* chooses the subsequence that was found first when the regular expression is matched. *Longest match* chooses the longest subsequence from the ones that match at that location. If there is more than one subsequence that has the maximal length, longest match chooses the one that was found first. For example, when first match is used, a search for the regular expression "b|bc" in the target sequence "abcd" matches the subsequence "b" because the left-hand term of the alternation matches that subsequence; therefore, first match does not try the right-hand term of the alternation. When longest match is used, the same search matches "bc" because "bc" is longer than "b".

A partial match succeeds if the match reaches the end of the target sequence without failing, even if it has not reached the end of the regular expression. Therefore, after a partial match succeeds, appending characters to the target sequence could cause a later partial match to fail. However, after a partial match fails, appending characters to the target sequence cannot cause a later partial match to succeed. For example, with a partial match, "ab" matches the target sequence "a" but not "ac".

Format Flags

ECMAScript FORMAT RULES	SED FORMAT RULES	REPLACEMENT TEXT
"\$&"	"&"	The character sequence that matches the entire regular expression (<code>[match[0].first, match[0].second)</code>)
"\$"		"\$"
	"\&"	"&"
"\$`" (dollar sign followed by back quote)		The character sequence that precedes the subsequence that matches the regular expression (<code>[match.prefix().first, match.prefix().second)</code>)

ECMAScript Format Rules	SED Format Rules	Replacement Text
"\$" (dollar sign followed by forward quote)		<p>The character sequence that follows the subsequence that matches the regular expression (</p> <pre>[match.suffix().first, match.suffix().second]</pre> <p>)</p>
"\$n"	"\n"	<p>The character sequence that matches the capture group at position <code>n</code>, where <code>n</code> is a number between 0 and 9 (</p> <pre>[match[n].first, match[n].second]</pre> <p>)</p>
	"\\n"	"\n"
"\$nn"		<p>The character sequence that matches the capture group at position <code>nn</code>, where <code>nn</code> is a number between 10 and 99 (</p> <pre>[match[nn].first, match[nn].second]</pre> <p>)</p>

See also

[C++ Standard Library Overview](#)

File System Navigation

10/31/2018 • 5 minutes to read • [Edit Online](#)

The `<filesystem>` header implements the C++ File System Technical Specification ISO/IEC TS 18822:2015 (Final draft: [ISO/IEC JTC 1/SC 22/WG 21 N4100](#)) and has types and functions that enable you to write platform-independent code for navigating the file system. Because it is cross-platform, it contains APIs that are not relevant for Windows systems. For example, this means that `is_fifo(const path&)` always returns **false** on Windows.

Overview

Use the `<filesystem>` APIs for the following tasks:

- iterate over files and directories under a specified path
- get information about files including the time created, size, extension, and root directory
- compose, decompose, and compare paths
- create, copy and delete directories
- copy and delete files

For more information about File IO using the Standard Library, see [iostream Programming](#).

Paths

Constructing and composing paths

Paths in Windows (since XP) are stored natively in Unicode. The `path` class automatically performs all necessary string conversions. It accepts arguments of both wide and narrow character arrays, as well as `std::string` and `std::wstring` types formatted as UTF8 or UTF16. The `path` class also automatically normalizes path separators. You can use a single forward slash as a directory separator in constructor arguments. This enables you to use the same strings to store paths in both Windows and UNIX environments:

```
path pathToDisplay(L"/FileSystemTest/SubDir3");    // OK!
path pathToDisplay2(L"\\FileSystemTest\\SubDir3"); // Still OK as always
path pathToDisplay3(LR"(\FileSystemTest\SubDir3)"); // Raw string literals are OK, too.
```

To concatenate two paths, you can use the overloaded `/` and `/=` operators, which are analogous to the `+` and `+=` operators on `std::string` and `std::wstring`. The `path` object will conveniently supply the separators if you don't.

```
path myRoot("C:/FileSystemTest"); // no trailing separator, no problem!
myRoot /= path("SubDirRoot");     // C:/FileSystemTest/SubDirRoot
```

Examining paths

The `path` class has several methods that return information about various parts of the path itself, as distinct from the file system entity it might refer to. You can get the root, the relative path, the file name, the file extension, and more. You can iterate over a path object to examine all the folders in the hierarchy. The following example shows how to iterate over a path (not the directory it refers to), and to retrieve information about its parts.

```

// filesystem_path_example.cpp
// compile by using: /EHsc
#include <string>
#include <iostream>
#include <sstream>
#include <filesystem>

using namespace std;
using namespace std::experimental::filesystem;

wstring DisplayPathInfo()
{
    // This path may or may not refer to an existing file. We are
    // examining this path string, not file system objects.
    path pathToDisplay(L"C:/FileSystemTest/SubDir3/SubDirLevel2/File2.txt ");

    wstringstream wos;
    int i = 0;
    wos << L"Displaying path info for: " << pathToDisplay << endl;
    for (path::iterator itr = pathToDisplay.begin(); itr != pathToDisplay.end(); ++itr)
    {
        wos << L"path part: " << i++ << L" = " << *itr << endl;
    }

    wos << L"root_name() = " << pathToDisplay.root_name() << endl
        << L"root_path() = " << pathToDisplay.root_path() << endl
        << L"relative_path() = " << pathToDisplay.relative_path() << endl
        << L"parent_path() = " << pathToDisplay.parent_path() << endl
        << L"filename() = " << pathToDisplay.filename() << endl
        << L"stem() = " << pathToDisplay.stem() << endl
        << L"extension() = " << pathToDisplay.extension() << endl;

    return wos.str();
}

void main(int argc, char* argv[])
{
    wcout << DisplayPathInfo() << endl;
    // wcout << ComparePaths() << endl; // see following example
    wcout << endl << L"Press Enter to exit" << endl;
    wstring input;
    getline(wcin, input);
}

```

The code produces this output:

```

Displaying path info for: C:\FileSystemTest\SubDir3\SubDirLevel2\File2.txt
path part: 0 = C:
path part: 1 = \
path part: 2 = FileSystemTest
path part: 3 = SubDir3
path part: 4 = SubDirLevel2
path part: 5 = File2.txt
root_name() = C:
root_path() = C:\
relative_path() = FileSystemTest\SubDir3\SubDirLevel2\File2.txt
parent_path() = C:\FileSystemTest\SubDir3\SubDirLevel2
filename() = File2.txt
stem() = File2
extension() = .txt

```

Comparing paths

The `path` class overloads the same comparison operators as `std::string` and `std::wstring`. When you compare two paths, you are performing a string comparison after the separators have been normalized. If a

trailing slash (or backslash) is missing it is not added and affects the comparison. The following example demonstrates how path values compare:

```
wstring ComparePaths()
{
    path p0(L"C:/Documents");           // no trailing separator
    path p1(L"C:/Documents/");          // p0 < p1
    path p2(L"C:/Documents/2013/");     // p1 < p2
    path p3(L"C:/Documents/2013/Reports/"); // p2 < p3
    path p4(L"C:/Documents/2014/");     // p3 < p4
    path p5(L"D:/Documents/2013/Reports/"); // p4 < p5

    wstringstream wos;
    wos << boolalpha <<
        p0.wstring() << L" < " << p1.wstring() << L": " << (p0 < p1) << endl <<
        p1.wstring() << L" < " << p2.wstring() << L": " << (p1 < p2) << endl <<
        p2.wstring() << L" < " << p3.wstring() << L": " << (p2 < p3) << endl <<
        p3.wstring() << L" < " << p4.wstring() << L": " << (p3 < p4) << endl <<
        p4.wstring() << L" < " << p5.wstring() << L": " << (p4 < p5) << endl;
    return wos.str();
}
```

```
C:\Documents < C:\Documents\: true
C:\Documents\ < C:\Documents\2013\: true
C:\Documents\2013\ < C:\Documents\2013\Reports\: true
C:\Documents\2013\Reports\ < C:\Documents\2014\: true
C:\Documents\2014\ < D:\Documents\2013\Reports\: true
```

To run this code, paste it into the full example above before `main` and uncomment the line that calls it in main.

Converting between path and string types

A `path` object is implicitly convertible to `std::wstring` or `std::string`. This means you can pass a path to functions such as [wofstream::open](#), as shown in this example:

```

// filesystem_path_conversion.cpp
// compile by using: /EHsc
#include <string>
#include <iostream>
#include <fstream>
#include <filesystem>

using namespace std;
using namespace std::experimental::filesystem;

void main(int argc, char* argv[])
{
    wchar_t* p = L"C:/Users/Public/Documents";
    path filePath(p);

    filePath /= L"NewFile.txt";

    // Open, write to, and close the file.
    wofstream writeFile(filePath, ios::out); // implicit conversion
    writeFile << L"Lorem ipsum\nDolor sit amet";
    writeFile.close();

    // Open, read, and close the file.
    wifstream readFile;
    wstring line;
    readFile.open(filePath); // implicit conversions
    wcout << L"File " << filePath << L" contains:" << endl;
    while (readFile.good())
    {
        getline(readFile, line);
        wcout << line << endl;
    }
    readFile.close();

    wcout << endl << L"Press Enter to exit" << endl;
    wstring input;
    getline(wcin, input);
}

```

```

File C:\Users\Public\Documents\NewFile.txt contains:
Lorem ipsum
Dolor sit amet

Press Enter to exit

```

Iterating directories and files

The `<filesystem>` header provides the [directory_iterator](#) type to iterate over single directories, and the [recursive_directory_iterator](#) class to iterate recursively over a directory and its subdirectories. After you construct an iterator by passing it a `path` object, the iterator points to the first `directory_entry` in the path. Create the end iterator by calling the default constructor.

When iterating through a directory, there are several kinds of items you might encounter, including but not limited to directories, files, symbolic links, and socket files. The `directory_iterator` returns its items as [directory_entry](#) objects.