std::map

Defined in header `<map>`

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees .

`std::map` meets the requirements of `Container`, `AllocatorAwareContainer`, `AssociativeContainer` and `ReversibleContainer`.

Member types

Member type	Definition
`key_type`	`Key`
`mapped_type`	`T`
`value_type`	`std::pair<const Key, T>`
`size_type`	Unsigned integral type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`key_compare`	`Compare`
`allocator_type`	`Allocator`
`reference`	`Allocator::reference` (until C++11) `value_type&` (since C++11)
`const_reference`	`Allocator::const_reference` (until C++11) `const value_type&` (since C++11)
`pointer`	`Allocator::pointer` (until C++11) `std::allocator_traits<Allocator>::pointer` (since C++11)
`const_pointer`	`Allocator::const_pointer` (until C++11) `std::allocator_traits<Allocator>::const_pointer` (since C++11)
`iterator`	`BidirectionalIterator`
`const_iterator`	Constant bidirectional iterator
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member classes

`value_compare`	compares objects of type `value_type` (class)

Member functions

(constructor)	constructs the `map` (public member function)
(destructor)	destructs the `map` (public member function)
`operator=`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Element access

`at` (C++11)	access specified element with bounds checking (public member function)

`operator[]`	access specified element (public member function)

Iterators

`begin` **`cbegin`**	returns an iterator to the beginning (public member function)
`end` **`cend`**	returns an iterator to the end (public member function)
`rbegin` **`crbegin`**	returns a reverse iterator to the beginning (public member function)
`rend` **`crend`**	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`insert_or_assign` (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`emplace_hint` (C++11)	constructs elements in-place using a hint (public member function)
`try_emplace` (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
`erase`	erases elements (public member function)
`swap`	swaps the contents (public member function)

Lookup

`count`	returns the number of elements matching specific key (public member function)
`find`	finds element with specific key (public member function)
`equal_range`	returns range of elements matching a specific key (public member function)
`lower_bound`	returns an iterator to the first element *not less* than the given key (public member function)
`upper_bound`	returns an iterator to the first element *greater* than the given key (public member function)

Observers

`key_comp`	returns the function that compares keys (public member function)
`value_comp`	returns the function that compares keys in objects of type value_type (public member function)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	lexicographically compares the values in the map (function template)
std::swap(std::map)	specializes the `std::swap` algorithm (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map&oldid=73713"

std::map::map

`explicit map( const Compare& comp = Compare(),`
`               const Allocator& alloc = Allocator() );`
	(1)	(until C++14)

`map() : map( Compare() ) {}`
`explicit map( const Compare& comp,`
`               const Allocator& alloc = Allocator() );`
		(since C++14)

`explicit map( const Allocator& alloc );`	(1)	(since C++11)

`template< class InputIterator >`
`map( InputIterator first, InputIterator last,`
`     const Compare& comp = Compare(),`
`     const Allocator& alloc = Allocator() );`
	(2)	

`template< class InputIterator >`
`map( InputIterator first, InputIterator last,`
`     const Allocator& alloc );`
		(since C++14)

`map( const map& other );`	(3)	
`map( const map& other, const Allocator& alloc );`	(3)	(since C++11)
`map( map&& other );`	(4)	(since C++11)
`map( map&& other, const Allocator& alloc );`	(4)	(since C++11)

`map( std::initializer_list<value_type> init,`
`     const Compare& comp = Compare(),`
`     const Allocator& alloc = Allocator() );`
	(5)	(since C++11)

`map( std::initializer_list<value_type> init,`
`     const Allocator& );`
		(since C++14)

Constructs new container from a variety of data sources and optionally using user supplied allocator `alloc` or comparison function object `comp`.

1) Default constructor. Constructs empty container.

2) Constructs the container with the contents of the range `[first, last)`.

3) Copy constructor. Constructs the container with the copy of the contents of `other`. If `alloc` is not provided, allocator is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())` .

4) Move constructor. Constructs the container with the contents of `other` using move semantics. If `alloc` is not provided, allocator is obtained by move-construction from the allocator belonging to `other`.

5) Constructs the container with the contents of the initializer list `init`.

Parameters

`alloc`	-	allocator to use for all memory allocations of this container
`comp`	-	comparison function object to use for all comparisons of keys
`first, last`	-	the range to copy the elements from
`other`	-	another container to be used as source to initialize the elements of the container with
`init`	-	initializer list to initialize the elements of the container with

Type requirements

- `InputIterator` must meet the requirements of `InputIterator`.
- `Compare` must meet the requirements of `Compare`.
- `Allocator` must meet the requirements of `Allocator`.

Complexity

1) Constant

2) *N log(N)* where `N = std::distance(first, last)` in general, linear in `N` if the range is already sorted by `value_comp()`.

3) Linear in size of `other`

4) Constant. If `alloc` is given and `alloc != other.get_allocator()` , then linear.

5) *N log(N)* where `N = init.size()` in general, linear in `N` if `init` is already sorted by `value_comp()`.

Example

Run this code

```
#include <iostream>
#include <string>
#include <iomanip>
#include <map>

template<typename Map>
void print_map(Map& m)
{
   std::cout << '{';
   for(auto& p: m)
        std::cout << p.first << ':' << p.second << ' ';
   std::cout << "}\n";
}

int main()
{
  // (1) Default constructor
  std::map<std::string, int> map1;
  map1["something"] = 69;
  map1["anything"] = 199;
  map1["that thing"] = 50;
  std::cout << "map1 = "; print_map(map1);

  // (2) Range constructor
  std::map<std::string, int> iter(map1.find("anything"), map1.end());
  std::cout << "\niter = "; print_map(iter);
  std::cout << "map1 = "; print_map(map1);

  // (3) Copy constructor
  std::map<std::string, int> copied(map1);
  std::cout << "\ncopied = "; print_map(copied);
  std::cout << "map1 = "; print_map(map1);

  // (4) Move constructor
  std::map<std::string, int> moved(std::move(map1));
  std::cout << "\nmoved = "; print_map(moved);
  std::cout << "map1 = "; print_map(map1);

  // (5) Initializer list constructor
  const std::map<std::string, int> init {
    {"this", 100},
    {"can", 100},
    {"be", 100},
    {"const", 100},
  };
  std::cout << "\ninit = "; print_map(init);
}
```

Output:

```
map1 = {anything:199 something:69 that thing:50 }

iter = {anything:199 something:69 that thing:50 }
map1 = {anything:199 something:69 that thing:50 }

copied = {anything:199 something:69 that thing:50 }
map1 = {anything:199 something:69 that thing:50 }

moved = {anything:199 something:69 that thing:50 }
map1 = {}

init = {be:100 can:100 const:100 this:100 }
```

See also

`operator=`	assigns values to the container (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/map&oldid=50563"

std::map::~map

```
~map();
```

Destructs the container. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.

Complexity

Linear in the size of the container.

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/%7Emap&oldid=50574"

std::map::operator=

`map& operator=( const map& other );`	(1)	
`map& operator=( map&& other );`	(2)	(since C++11)
`map& operator=( std::initializer_list<value_type> ilist );`	(3)	(since C++11)

Replaces the contents of the container.

1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. If `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If the target and the source allocators do not compare equal, the target (`*this`) allocator is used to deallocate the memory, then `other`'s allocator is used to allocate it before copying the elements. (since C++11)

2) Move assignment operator. Replaces the contents with those of `other` using move semantics (i.e. the data in `other` is moved from `other` into this container). `other` is in a valid but unspecified state afterwards. If `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If it is `false` and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must move-assign each element individually, allocating additional memory using its own allocator as needed.

3) Replaces the contents with those identified by initializer list `ilist`.

Parameters

other	-	another container to use as data source
ilist	-	initializer list to use as data source

Return value

`*this`

Complexity

1) Linear in the size of the `other`.

2) Constant unless `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is false and the allocators do not compare equal (in which case linear).

3) Linear in the size of `ilist`.

Exceptions

(since C++17)

2) `noexcept` specification:

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value
&& std::is_nothrow_move_assignable<Compare>::value)
```

Example

The following code uses to assign one `std::map` to another:

Run this code

```
#include <map>
#include <iostream>

void display_sizes(const std::map<int, int> &nums1,
                   const std::map<int, int> &nums2,
                   const std::map<int, int> &nums3)
{
    std::cout << "nums1: " << nums1.size()
              << " nums2: " << nums2.size()
              << " nums3: " << nums3.size() << '\n';
}

int main()
{
    std::map<int, int> nums1 {{3, 1}, {4, 1}, {5, 9},
                              {6, 1}, {7, 1}, {8, 9}};
    std::map<int, int> nums2;
    std::map<int, int> nums3;

    std::cout << "Initially:\n";
    display_sizes(nums1, nums2, nums3);

    // copy assignment copies data from nums1 to nums2
    nums2 = nums1;

    std::cout << "After assigment:\n";
    display_sizes(nums1, nums2, nums3);

    // move assignment moves data from nums1 to nums3,
    // modifying both nums1 and nums3
    nums3 = std::move(nums1);

    std::cout << "After move assigment:\n";
    display_sizes(nums1, nums2, nums3);
}
```

Output:

```
Initially:
nums1: 6 nums2: 0 nums3: 0
After assigment:
nums1: 6 nums2: 6 nums3: 0
After move assigment:
nums1: 0 nums2: 6 nums3: 6
```

See also

(constructor)	constructs the `map` (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/operator%3D&oldid=43411"

std::map::get_allocator

```
allocator_type get_allocator() const;
```

Returns the allocator associated with the container.

Parameters

(none)

Return value

The associated allocator.

Complexity

Constant.

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/get_allocator&oldid=50560"

std::map::at

`T& at( const Key& key );`	(1)	(since C++11)
`const T& at( const Key& key ) const;`	(2)	(since C++11)

Returns a reference to the mapped value of the element with key equivalent to `key`. If no such element exists, an exception of type `std::out_of_range` is thrown.

Parameters

key	-	the key of the element to find

Return value

Reference to the mapped value of the requested element

Exceptions

`std::out_of_range` if the container does not have an element with the specified `key`

Complexity

Logarithmic in the size of the container.

See also

operator[]	access specified element (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/at&oldid=50549"

std::map::operator[]

`T& operator[]( const Key& key );`	(1)	
`T& operator[]( Key&& key );`	(2)	(since C++11)

Returns a reference to the value that is mapped to a key equivalent to `key`, performing an insertion if such key does not already exist.

If an insertion is performed, the mapped value is value-initialized (default-constructed for class types, zero-initialized otherwise) and a reference to it is returned.

1) Inserts `value_type(key, T())`
- `key_type` must meet the requirements of `CopyConstructible`.
- `mapped_type` must meet the requirements of `DefaultConstructible`. (since C++11)

2) Inserts `value_type(std::move(key), T())`
- `key_type` must meet the requirements of `MoveConstructible`. (since C++11)
- `mapped_type` must meet the requirements of `DefaultConstructible`. (since C++11)

No iterators or references are invalidated.

Parameters

`key` - the key of the element to find

Return value

Reference to the mapped value of the new element if no element with key `key` existed. Otherwise a reference to the mapped value of the existing element whose key is equivalent to `key`.

Exceptions

If an exception is thrown by any operation, the insertion has no effect

Complexity

Logarithmic in the size of the container.

Notes

Until C++11, the overload (1) was specified to be equivalent to `(insert(std::make_pair(key, T())).first)->second` , which required T to be CopyConstructible.

Example

This example demonstrates how to modify existing values and insert new values using `operator[]`:

Run this code

```
#include <iostream>
#include <map>

int main()
{
    std::map<char, int> letter_counts {{'a', 27}, {'b', 3}, {'c', 1}};

    std::cout << "initially:\n";
    for (const auto &pair : letter_counts) {
        std::cout << pair.first << ": " << pair.second << '\n';
    }
    letter_counts['b'] = 42;  // update an existing value

    letter_counts['x'] = 9;  // insert a new value

    std::cout << "after modifications:\n";
    for (const auto &pair : letter_counts) {
        std::cout << pair.first << ": " << pair.second << '\n';
    }
}
```

Output:

```
initially:
a: 27
b: 3
c: 1
after modifications:
a: 27
b: 42
c: 1
x: 9
```

The following example counts the occurrences of each word in a vector of strings:

Run this code

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

int main()
{
    std::vector<std::string> words = {
        "this", "sentence", "is", "not", "a", "sentence",
        "this", "sentence", "is", "a", "hoax"
    };

    std::map<std::string, size_t>  word_map;
    for (const auto &w : words) {
        ++word_map[w];
    }

    for (const auto &pair : word_map) {
        std::cout << pair.second
                  << " occurrences of word '"
                  << pair.first << "'\n";
    }
}
```

Output:

```
1 occurrences of word 'hoax'
2 occurrences of word 'this'
2 occurrences of word 'a'
2 occurrences of word 'is'
1 occurrences of word 'not'
3 occurrences of word 'sentence'
```

See also

at (C++11)	access specified element with bounds checking (public member function)
insert_or_assign (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)

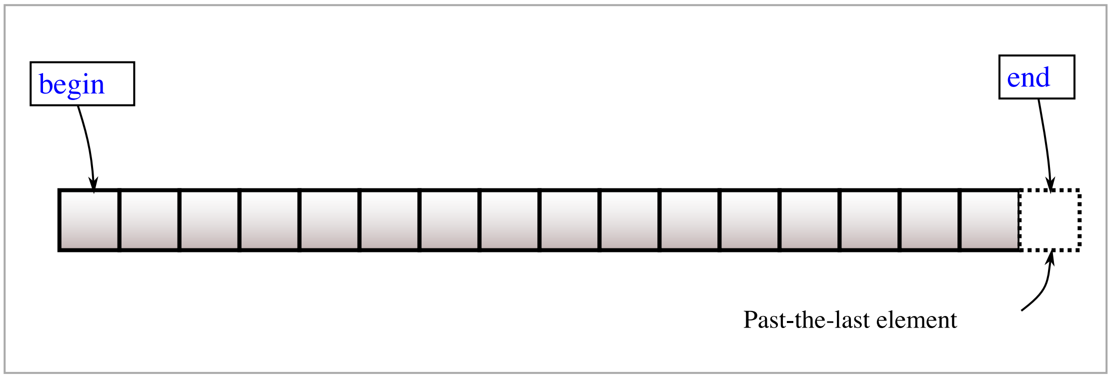
Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/operator_at&oldid=73773"

std::map::begin, std::map::cbegin

`iterator begin();`	
`const_iterator begin() const;`	
`const_iterator cbegin() const;`	(since C++11)

Returns an iterator to the first element of the container.

If the container is empty, the returned iterator will be equal to `end()`.



Parameters

(none)

Return value

Iterator to the first element

Exceptions

(none)	(until C++11)
`noexcept` specification: `noexcept`	(since C++11)

Complexity

Constant

Example

This section is incomplete
Reason: no example

See also

end **cend**	returns an iterator to the end (public member function)

Example

Run this code

```
#include <map>
#include <string>
```

```
#include <iostream>
#include <iterator>

int main() {
  std::map<std::string,std::string> a_map;
  a_map["Geely"]    = "Chinese";
  a_map["Peugeot"]  = "French";
  a_map["Mercedes"] = "German";
  a_map["Toyota"]   = "Japanese";
  a_map["Ford"]     = "American";
  a_map["Fiat"]     = "Italian";

  for (auto it = a_map.cbegin(); it != std::next(a_map.cbegin(), 3); ++it) {
    std::cout << it->first << " : " << it->second << '\n';
  }
}
```

Output:

```
Fiat : Italian
Ford : American
Geely : Chinese
```

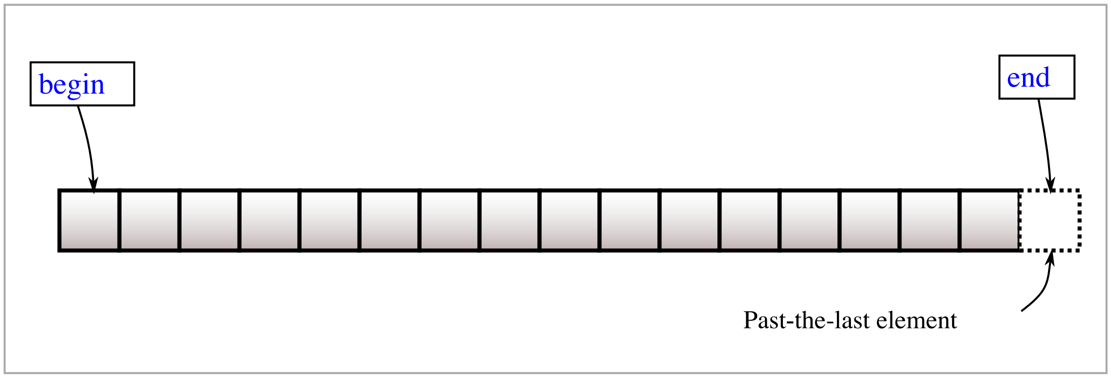
Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/begin&oldid=72255"

std::map::end, std::map::cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;     (since C++11)
```

Returns an iterator to the element following the last element of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Iterator to the element following the last element.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

See also

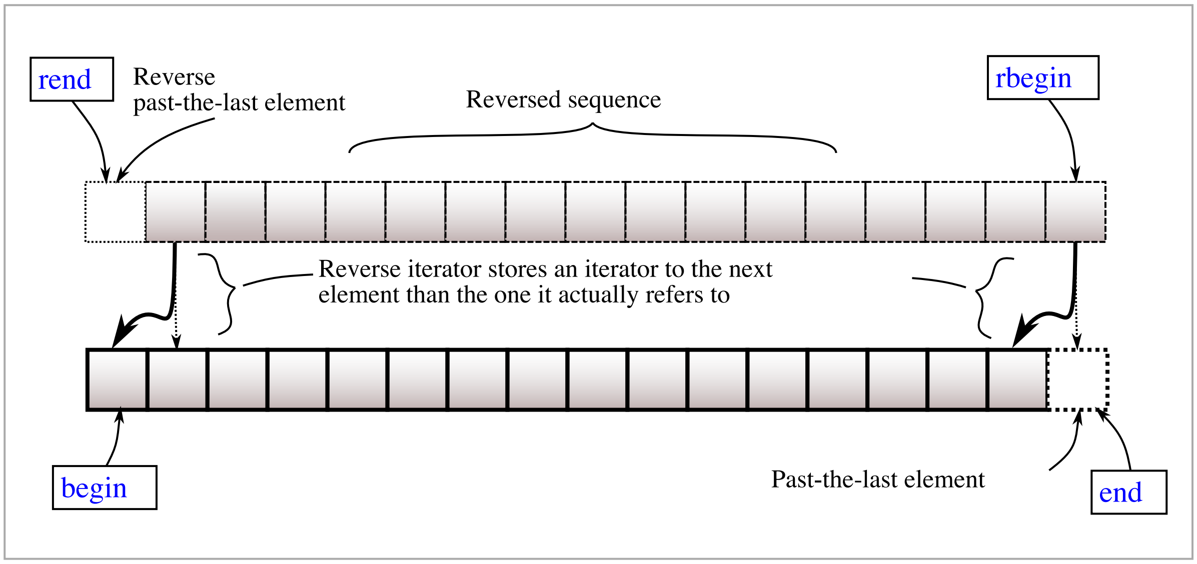
begin **cbegin**	returns an iterator to the beginning (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/end&oldid=50556"

std::map::**rbegin**, std::map::**crbegin**

`reverse_iterator rbegin();`	
`const_reverse_iterator rbegin() const;`	
`const_reverse_iterator crbegin() const;`	(since C++11)

Returns a reverse iterator to the first element of the reversed container. It corresponds to the last element of the non-reversed container.


rend
Reverse
past-the-last element
Reversed sequence
rbegin
Reverse iterator stores an iterator to the next
element than the one it actually refers to
begin
Past-the-last element
end

Parameters

(none)

Return value

Reverse iterator to the first element.

Exceptions

(none)	(until C++11)
`noexcept` specification: `noexcept`	(since C++11)

Complexity

Constant.

See also

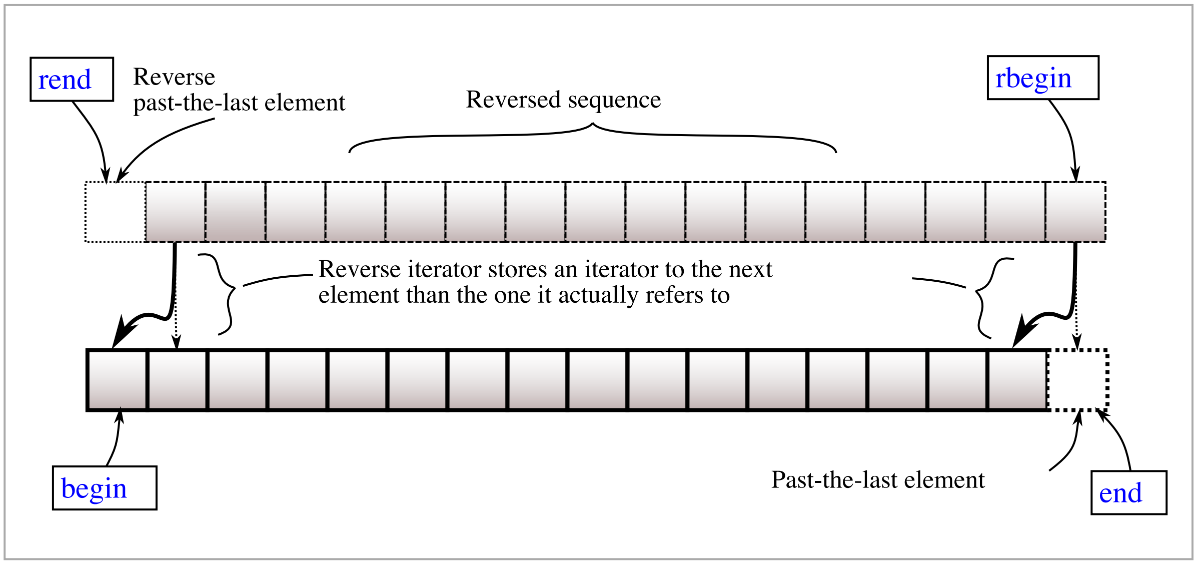
`rend` **`crend`**	returns a reverse iterator to the end (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/rbegin&oldid=50566"

std::map::rend, std::map::crend

```
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;        (since C++11)
```

Returns a reverse iterator to the element following the last element of the reversed container. It corresponds to the element preceding the first element of the non-reversed container. This element acts as a placeholder, attempting to access it results in undefined behavior.


rend
Reverse
past-the-last element
Reversed sequence
rbegin
Reverse iterator stores an iterator to the next
element than the one it actually refers to
begin
Past-the-last element
end

Parameters

(none)

Return value

Reverse iterator to the element following the last element.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

See also

rbegin **crbegin**	returns a reverse iterator to the beginning (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/rend&oldid=50567"

std::map::empty

```
bool empty() const;
```

Checks if the container has no elements, i.e. whether `begin() == end()`.

Parameters

(none)

Return value

`true` if the container is empty, `false` otherwise

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

Example

The following code uses `empty` to check if a `std::map<int, int>` contains any elements:

Run this code

```
#include <map>
#include <iostream>
#include <utility>

int main()
{
    std::map<int,int> numbers;
    std::cout << "Initially, numbers.empty(): " << numbers.empty() << '\n';

    numbers.emplace(42, 13);
    numbers.insert(std::make_pair(13317, 123));
    std::cout << "After adding elements, numbers.empty(): " << numbers.empty() << '\n';
}
```

Output:

```
Initially, numbers.empty(): 1
After adding elements, numbers.empty(): 0
```

See also

size	returns the number of elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/empty&oldid=50555"

std::map::size

```
size_type size() const;
```

Returns the number of elements in the container, i.e. `std::distance(begin(), end())`.

Parameters

(none)

Return value

The number of elements in the container.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

Example

The following code uses `size` to display the number of elements in a `std::map`:

Run this code

```
#include <map>
#include <iostream>

int main()
{
    std::map<int,char> nums {{1, 'a'}, {3, 'b'}, {5, 'c'}, {7, 'd'}};

    std::cout << "nums contains " << nums.size() << " elements.\n";
}
```

Output:

```
nums contains 4 elements.
```

See also

`empty`	checks whether the container is empty (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/size&oldid=50568"

std::map::max_size

```
size_type max_size() const;
```

Returns the maximum number of elements the container is able to hold due to system or library implementation limitations, i.e. `std::distance(begin(), end())` for the largest container.

Parameters

(none)

Return value

Maximum number of elements.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

Notes

This value is typically equal to `std::numeric_limits<size_type>::max()`, and reflects the theoretical limit on the size of the container. At runtime, the size of the container may be limited to a value smaller than `max_size()` by the amount of RAM available.

Example

Run this code

```
#include <iostream>
#include <map>

int main()
{
    std::map<char,char> s;
    std::cout << "Maximum size of a 'map' is " << s.max_size() << "\n";
}
```

Possible output:

```
Maximum size of a 'map' is 18446744073709551615
```

See also

size	returns the number of elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/max_size&oldid=50564"

std::map::clear

```
void clear();
```

Removes all elements from the container.

Invalidates any references, pointers, or iterators referring to contained elements. May invalidate any past-the-end iterators.

Parameters

(none)

Return value

(none)

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Linear in the size of the container.

See also

`erase`	erases elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/clear&oldid=50551"

std::map::insert

`std::pair<iterator,bool> insert( const value_type& value );`	(1)	

`template< class P >`
`std::pair<iterator,bool> insert( P&& value );`
	(2)	(since C++11)

`std::pair<iterator,bool> insert( value_type&& value );`	(2)	(since C++17)
`iterator insert( iterator hint, const value_type& value );`	(3)	(until C++11)
`iterator insert( const_iterator hint, const value_type& value );`		(since C++11)

`template< class P >`
`iterator insert( const_iterator hint, P&& value );`
	(4)	(since C++17)

`iterator insert( const_iterator hint, value_type&& value );`	(4)	(since C++11)

`template< class InputIt >`
`void insert( InputIt first, InputIt last );`
	(5)	

`void insert( std::initializer_list<value_type> ilist );`	(6)	(since C++11)

Inserts element(s) into the container, if the container doesn't already contain an element with an equivalent key.

1-2) Inserts `value`. The overload (2) is equivalent to `emplace(std::forward<P>(value))` and only participates in overload resolution if `std::is_constructible<value_type, P&&>::value == true`.

3-4) Inserts `value` in the position as close as possible, just prior(since C++11), to `hint`. The overload (4) is equivalent to `emplace_hint(hint, std::forward<P>(value))` and only participates in overload resolution if `std::is_constructible<value_type, P&&>::value == true`.

5) Inserts elements from range `[first, last)`.

6) Inserts elements from initializer list `ilist`.

No iterators or references are invalidated.

Parameters

`hint`	-	iterator, used as a suggestion as to where to start the search (until C++11) iterator to the position before which the new element will be inserted (since C++11)
`value`	-	element value to insert
`first, last`	-	range of elements to insert
`ilist`	-	initializer list to insert the values from

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.

Return value

1-2) Returns a pair consisting of an iterator to the inserted element (or to the element that prevented the insertion) and a `bool` denoting whether the insertion took place.

3-4) Returns an iterator to the inserted element, or to the element that prevented the insertion.

5-6) (none)

Exceptions

1-4) If an exception is thrown by any operation, the insertion has no effect.

This section is incomplete
Reason: cases 5-6

Complexity

1-2) Logarithmic in the size of the container, `O(log(size()))`.

3-4) Amortized constant if the insertion happens in the position just *after* the hint, logarithmic in the size of the container otherwise.	(until C++11)
3-4) Amortized constant if the insertion happens in the position just *before* the hint, logarithmic in the size of the container otherwise.	(since C++11)

5-6) `O(N*log(size() + N))`, where N is the number of elements to insert.

See also

`emplace` (C++11)	constructs element in-place (public member function)
`emplace_hint` (C++11)	constructs elements in-place using a hint (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/insert&oldid=79461"

std::map::insert_or_assign

`template <class M>`
`pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);`
	(1)	(since C++17)

`template <class M>`
`pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);`
	(2)	(since C++17)

`template <class M>`
`iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);`
	(3)	(since C++17)

`template <class M>`
`iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);`
	(4)	(since C++17)

1,3) If a key equivalent to `k` already exists in the container, assigns `std::forward<M>(obj)` to the `mapped_type` corresponding to the key `k`. If the key does not exist, inserts the new value as if by insert, constructing it from `value_type(k, std::forward<M>(obj))`

2,4) Same as (1,3), except the mapped value is constructed from `value_type(std::move(k), std::forward<M>(obj))`

No iterators or references are invalidated.

Parameters

k	-	the key used both to look up and to insert if not found
hint	-	iterator to the position before which the new element will be inserted
args	-	arguments to forward to the constructor of the element

Return value

1,2) The bool component is `true` if the insertion took place and `false` if the assignment took place. The iterator component is pointing at the element that was inserted or updated

3,4) Iterator pointing at the element that was inserted or updated

Complexity

1,2) Same as for emplace

3,4) Same as for emplace_hint

Notes

`insert_or_assign` returns more information than `operator[]` and does not require default-constructibility of the mapped type.

Example

This section is incomplete
Reason: no example

See also

operator[]	access specified element (public member function)
at (C++11)	access specified element with bounds checking (public member function)
insert	inserts elements (public member function)
emplace (C++11)	constructs element in-place (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/insert_or_assign&oldid=74463"

std::map::emplace

```
template< class... Args >
std::pair<iterator,bool> emplace( Args&&... args );
```

(since C++11)

Inserts a new element into the container by constructing it in-place with the given `args` if there is no element with the key in the container.

Careful use of `emplace` allows the new element to be constructed while avoiding unnecessary copy or move operations. The constructor of the new element (i.e. `std::pair<const Key, T>`) is called with exactly the same arguments as supplied to `emplace`, forwarded via `std::forward<Args>(args)...`.

No iterators or references are invalidated.

Parameters

`args` - arguments to forward to the constructor of the element

Return value

Returns a pair consisting of an iterator to the inserted element, or the already-existing element if no insertion happened, and a `bool` denoting whether the insertion took place.

Exceptions

If an exception is thrown by any operation, this function has no effect.

Complexity

Logarithmic in the size of the container.

Example

Run this code

```
#include <iostream>
#include <utility>
#include <string>

#include <map>
int main()
{
    std::map<std::string, std::string> m;

    // uses pair's move constructor
    m.emplace(std::make_pair(std::string("a"), std::string("a")));

    // uses pair's converting move constructor
    m.emplace(std::make_pair("b", "abcd"));

    // uses pair's template constructor
    m.emplace("d", "ddd");

    // uses pair's piecewise constructor
    m.emplace(std::piecewise_construct,
              std::forward_as_tuple("c"),
              std::forward_as_tuple(10, 'c'));

    for (const auto &p : m) {
        std::cout << p.first << " => " << p.second << '\n';
    }
}
```

Output:

```
a => a
b => abcd
c => cccccccccc
d => ddd
```

See also

emplace_hint (C++11)	constructs elements in-place using a hint (public member function)
try_emplace (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
insert	inserts elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/emplace&oldid=50553"

std::map::emplace_hint

```
template <class... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

(since C++11)

Inserts a new element to the container as close as possible to the position just before `hint`. The element is constructed in-place, i.e. no copy or move operations are performed.

The constructor of the element type (`value_type`, that is, `std::pair<const Key, T>`) is called with exactly the same arguments as supplied to the function, forwarded with `std::forward<Args>(args)...` .

No iterators or references are invalidated.

Parameters

hint	-	iterator to the position before which the new element will be inserted
args	-	arguments to forward to the constructor of the element

Return value

Returns an iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the already existing element with the equivalent key.

Complexity

Logarithmic in the size of the container in general, but amortized constant if the new element is inserted just before `hint`.

See also

emplace (C++11)	constructs element in-place (public member function)
insert	inserts elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/emplace_hint&oldid=73438"

std::map::try_emplace

`template <class... Args>`
`pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);`
	(1)	(since C++17)

`template <class... Args>`
`pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);`
	(2)	(since C++17)

`template <class... Args>`
`iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);`
	(3)	(since C++17)

`template <class... Args>`
`iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);`
	(4)	(since C++17)

1) If a key equivalent to `k` already exists in the container, does nothing. Otherwise, behaves like emplace except that the element is constructed as `value_type(std::piecewise_construct, std::forward_as_tuple(k), std::forward_as_tuple(forward<Args>(args)...))`

2) If a key equivalent to `k` already exists in the container, does nothing. Otherwise, behaves like emplace except that the element is constructed as `value_type(std::piecewise_construct, std::forward_as_tuple(std::move(k)), std::forward_as_tuple(forward<Args>(args)...))`

3) If a key equivalent to `k` already exists in the container, does nothing. Otherwise, behaves like emplace_hint except that the element is constructed as `value_type(std::piecewise_construct, std::forward_as_tuple(k), std::forward_as_tuple(forward<Args>(args)...))`

4) If a key equivalent to `k` already exists in the container, does nothing. Otherwise, behaves like emplace_hint except that the element is constructed as `value_type(std::piecewise_construct, std::forward_as_tuple(std::move(k)), std::forward_as_tuple(forward<Args>(args)...))`

No iterators or references are invalidated.

Parameters

k	-	the key used both to look up and to insert if not found
hint	-	iterator to the position before which the new element will be inserted
args	-	arguments to forward to the constructor of the element

Return value

1,2) Same as for emplace

3,4) Same as for emplace_hint

Complexity

1,2) Same as for emplace

3,4) Same as for emplace_hint

Notes

Unlike insert or emplace, these functions do not steal from move-only arguments if the insertion does not happen, which makes it easy to manipulate maps whose values are move-only types, such as `std::map<std::string, std::unique_ptr<foo>>`. In addition, `try_emplace` treats the key and the arguments to the `mapped_type` separately, unlike emplace, which requires the arguments to construct a `value_type` (that is, a `std::pair`)

Example

This section is incomplete
Reason: no example

See also

emplace (C++11)	constructs element in-place (public member function)
emplace_hint (C++11)	constructs elements in-place using a hint (public member function)
insert	inserts elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/try_emplace&oldid=74454"

std::map::erase

`void erase( iterator pos );`		(until C++11)
`iterator erase( iterator pos );`	(1)	(since C++17)
`iterator erase( const_iterator pos );`		(since C++11)
`void erase( iterator first, iterator last );`	(2)	(until C++11)
`iterator erase( const_iterator first, const_iterator last );`		(since C++11)
`size_type erase( const key_type& key );`	(3)	

Removes specified elements from the container.

1) Removes the element at `pos`.

2) Removes the elements in the range `[first; last)`, which must be a valid range in `*this`.

3) Removes the element (if one exists) with the key equivalent to `key`.

References and iterators to the erased elements are invalidated. Other references and iterators are not affected.

The iterator `pos` must be valid and dereferenceable. Thus the `end()` iterator (which is valid, but is not dereferencable) cannot be used as a value for `pos`.

Parameters

pos	-	iterator to the element to remove
first, last	-	range of elements to remove
key	-	key value of the elements to remove

Return value

1-2) Iterator following the last removed element.

3) Number of elements removed.

Exceptions

1,2) (none)

3) Any exceptions thrown by the `Compare` object.

Complexity

Given an instance `c` of `map`:

1) Amortized constant

2) `log(c.size()) + std::distance(first, last)`

3) `log(c.size()) + c.count(k)`

Example

Run this code

```
#include <map>
#include <iostream>
int main()
{
    std::map<int, std::string> c = {{1, "one"}, {2, "two"}, {3, "three"},
                                    {4, "four"}, {5, "five"}, {6, "six"}};
    // erase all odd numbers from c
    for(auto it = c.begin(); it != c.end(); )
        if(it->first % 2 == 1)
            it = c.erase(it);
        else
            ++it;
    for(auto& p : c)
        std::cout << p.second << ' ';
}
```

Output:

```
two four six
```

See also

`clear`	clears the contents (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/erase&oldid=50558"

std::map::swap

```
void swap( map& other );
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual elements.

All iterators and references remain valid. The past-the-end iterator is invalidated.

The `Pred` objects must be `Swappable`, and they are exchanged using unqualified call to non-member `swap`.

If `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true, then the allocators are exchanged using an unqualified call to non-member `swap`. Otherwise, they are not swapped (and if `get_allocator() != other.get_allocator()`, the behavior is undefined).	(since C++11)

Parameters

other - container to exchange the contents with

Return value

(none)

Exceptions

Any exception thrown by the swap of the `Compare` objects.	(until C++17)
`noexcept` specification: `noexcept(std::allocator_traits<Allocator>::is_always_equal::value && noexcept(std::swap(std::declval<Compare&>(),std::declval<Compare&>())))`	(since C++17)

Complexity

Constant.

See also

std::swap(std::map)	specializes the `std::swap` algorithm (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/swap&oldid=50569"

std::map::count

`size_type count( const Key& key ) const;`	(1)	

`template< class K >`
`size_type count( const K& x ) const;`
	(2)	(since C++14)

1) Returns the number of elements with key `key`, which is either 1 or 0 since this container does not allow duplicates

2) Returns the number of elements with key that compares *equivalent* to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. They allow calling this function without constructing an instance of `Key`.

Parameters

key	-	key value of the elements to count
x	-	alternative value to compare to the keys

Return value

Number of elements with key `key`, that is either 1 or 0

Complexity

Logarithmic in the size of the container.

See also

find	finds element with specific key (public member function)
equal_range	returns range of elements matching a specific key (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/count&oldid=65126"

std::map::find

`iterator find( const Key& key );`	(1)	
`const_iterator find( const Key& key ) const;`	(2)	
`template< class K > iterator find( const K& x );`	(3)	(since C++14)
`template< class K > const_iterator find( const K& x ) const;`	(4)	(since C++14)

1,2) Finds an element with key equivalent to `key`.

3,4) Finds an element with key that compares *equivalent* to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. It allows calling this function without constructing an instance of `Key`

Parameters

key	-	key value of the element to search for
x	-	a value of any type that can be transparently compared with a key

Return value

Iterator to an element with key equivalent to `key`. If no such element is found, past-the-end (see `end()`) iterator is returned.

Complexity

Logarithmic in the size of the container.

Example

Run this code

```
#include <iostream>
#include <map>

int main()
{
    std::map<int,char> example = {{1,'a'},{2,'b'}};

    auto search = example.find(2);
    if(search != example.end()) {
        std::cout << "Found " << search->first << " " << search->second << '\n';
    }
    else {
        std::cout << "Not found\n";
    }
}
```

Output:

```
Found 2 b
```

See also

count	returns the number of elements matching specific key (public member function)
equal_range	returns range of elements matching a specific key (public member function)

Example

Demonstrates the risk of accessing non-existing elements via operator [].

Run this code

```
#include <string>
#include <iostream>
#include <map>

int main()
{
    std::map<std::string,int> my_map;
    my_map["x"] =  11;
    my_map["y"] = 23;

    auto it = my_map.find("x");
    if (it != my_map.end()) std::cout << "x: " << it->second << "\n";

    it = my_map.find("z");
    if (it != my_map.end()) std::cout << "z1: " << it->second << "\n";

    // Accessing a non-existing element creates it
    if (my_map["z"] == 42) std::cout << "Oha!\n";

    it = my_map.find("z");
    if (it != my_map.end()) std::cout << "z2: " << it->second << "\n";
}
```

Output:

```
x: 11
z2: 0
```

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/find&oldid=65563"

std::map::equal_range

`std::pair<iterator,iterator> equal_range( const Key& key );`	(1)	
`std::pair<const_iterator,const_iterator> equal_range( const Key& key ) const;`	(2)	

`template< class K >`
`std::pair<iterator,iterator> equal_range( const K& x );`
	(3)	(since C++14)

`template< class K >`
`std::pair<const_iterator,const_iterator> equal_range( const K& x ) const;`
	(4)	(since C++14)

Returns a range containing all elements with the given key in the container. The range is defined by two iterators, one pointing to the first element that is *not less* than `key` and another pointing to the first element *greater* than `key`. Alternatively, the first iterator may be obtained with `lower_bound()`, and the second with `upper_bound()`.

1,2) Compares the keys to `key`.

3,4) Compares the keys to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. They allow calling this function without constructing an instance of `Key`.

> This section is incomplete
> Reason: explain better

Parameters

key	-	key value to compare the elements to
x	-	alternative value that can be compared to `Key`

Return value

`std::pair` containing a pair of iterators defining the wanted range: the first pointing to the first element that is not *less* than `key` and the second pointing to the first element *greater* than `key`.

If there are no elements *not less* than `key`, past-the-end (see `end()`) iterator is returned as the first element. Similarly if there are no elements *greater* than `key`, past-the-end iterator is returned as the second element.

Complexity

Logarithmic in the size of the container.

Example

> This section is incomplete
> Reason: no example

See also

find	finds element with specific key (public member function)
upper_bound	returns an iterator to the first element *greater* than the given key (public member function)
lower_bound	returns an iterator to the first element *not less* than the given key (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/equal_range&oldid=65093"

std::map::lower_bound

`iterator lower_bound( const Key& key );`	(1)	
`const_iterator lower_bound( const Key& key ) const;`	(1)	

`template< class K >`
`iterator lower_bound(const K& x);`
	(2)	(since C++14)

`template< class K >`
`const_iterator lower_bound(const K& x) const;`
	(2)	(since C++14)

1) Returns an iterator pointing to the first element that is *not less* than `key`.

2) Returns an iterator pointing to the first element that compares *not less* to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. They allow calling this function without constructing an instance of `Key`.

Parameters

key	-	key value to compare the elements to
x	-	alternative value that can be compared to `Key`

Return value

Iterator pointing to the first element that is not *less* than `key`. If no such element is found, a past-the-end iterator (see `end()`) is returned.

Complexity

Logarithmic in the size of the container.

See also

equal_range	returns range of elements matching a specific key (public member function)
upper_bound	returns an iterator to the first element *greater* than the given key (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/lower_bound&oldid=50562"

std::map::upper_bound

`iterator upper_bound( const Key& key );`	(1)	
`const_iterator upper_bound( const Key& key ) const;`	(1)	

`template< class K >`
`iterator upper_bound( const K& x );`
	(2)	(since C++14)

`template< class K >`
`const_iterator upper_bound( const K& x ) const;`
	(2)	(since C++14)

1) Returns an iterator pointing to the first element that is *greater* than `key`.

2) Returns an iterator pointing to the first element that compares *greater* to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. They allow calling this function without constructing an instance of `Key`.

Parameters

key	-	key value to compare the elements to
x	-	alternative value that can be compared to `Key`

Return value

Iterator pointing to the first element that is *greater* than `key`. If no such element is found, past-the-end (see `end()`) iterator is returned.

Complexity

Logarithmic in the size of the container.

See also

equal_range	returns range of elements matching a specific key (public member function)
lower_bound	returns an iterator to the first element *not less* than the given key (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/upper_bound&oldid=50571"

std::map::key_comp

```
key_compare key_comp() const;
```

Returns the function object that compares the keys, which is a copy of this container's constructor argument `comp`.

Parameters

(none)

Return value

The key comparison function object.

Complexity

Constant.

See also

`value_comp`	returns the function that compares keys in objects of type value_type (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/key_comp&oldid=50561"

std::map::value_comp

```
std::map::value_compare value_comp() const;
```

Returns a function object that compares objects of type std::map::value_type (key-value pairs) by using `key_comp` to compare the first components of the pairs.

Parameters

(none)

Return value

The value comparison function object.

Complexity

Constant.

See also

key_comp	returns the function that compares keys (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/value_comp&oldid=50572"

std::map::value_compare

```
class value_compare;
```

`std::map::value_compare` is a function object that compares objects of type `std::map::value_type` (key-value pairs) by comparing of the first components of the pairs.

Member types

Type	Definition
`result_type`	`bool`
`first_argument_type`	`value_type`
`second_argument_type`	`value_type`

Protected member objects

`Compare` **`comp`**	the stored comparator (protected member object)

Member functions

(constructor)	constructs a new `value_compare` object (protected member function)
`operator()`	compares two values of type `value_type` (public member function)

std::map<Key,T,Compare,Alloc>::value_compare::value_compare

```
protected:
value_compare( Compare c );
```

Initializes the internal instance of the comparator to `c`.

Parameters

`c` - comparator to assign

std::map<Key,T,Compare,Alloc>::value_compare::operator()

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored comparator.

Parameters

`lhs, rhs` - values to compare

Return value

`comp(lhs.first, rhs.first)`.

Exceptions

(none)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/value_compare&oldid=50573"

operator==,!=,<,<=,>,>=(std::map)

```
template< class Key, class T, class Compare, class Alloc >
bool operator==( const map<Key,T,Compare,Alloc>& lhs,                 (1)
                 const map<Key,T,Compare,Alloc>& rhs );
```

```
template< class Key, class T, class Compare, class Alloc >
bool operator!=( const map<Key,T,Compare,Alloc>& lhs,                 (2)
                 const map<Key,T,Compare,Alloc>& rhs );
```

```
template< class Key, class T, class Compare, class Alloc >
bool operator<( const map<Key,T,Compare,Alloc>& lhs,                  (3)
                const map<Key,T,Compare,Alloc>& rhs );
```

```
template< class Key, class T, class Compare, class Alloc >
bool operator<=( const map<Key,T,Compare,Alloc>& lhs,                 (4)
                 const map<Key,T,Compare,Alloc>& rhs );
```

```
template< class Key, class T, class Compare, class Alloc >
bool operator>( const map<Key,T,Compare,Alloc>& lhs,                  (5)
                const map<Key,T,Compare,Alloc>& rhs );
```

```
template< class Key, class T, class Compare, class Alloc >
bool operator>=( const map<Key,T,Compare,Alloc>& lhs,                 (6)
                 const map<Key,T,Compare,Alloc>& rhs );
```

Compares the contents of two containers.

1-2) Checks if the contents of `lhs` and `rhs` are equal, that is, whether `lhs.size() == rhs.size()` and each element in `lhs` compares equal with the element in `rhs` at the same position.

3-6) Compares the contents of `lhs` and `rhs` lexicographically. The comparison is performed by a function equivalent to `std::lexicographical_compare`.

Parameters

`lhs, rhs` - containers whose contents to compare

- `T, Key` must meet the requirements of `EqualityComparable` in order to use overloads (1-2).

- `Key` must meet the requirements of `LessThanComparable` in order to use overloads (3-6). The ordering relation must establish total order.

Return value

1) `true` if the contents of the containers are equal, `false` otherwise

2) `true` if the contents of the containers are not equal, `false` otherwise

3) `true` if the contents of the `lhs` are lexicographically *less* than the contents of `rhs`, `false` otherwise

4) `true` if the contents of the `lhs` are lexicographically *less* than or *equal* the contents of `rhs`, `false` otherwise

5) `true` if the contents of the `lhs` are lexicographically *greater* than the contents of `rhs`, `false` otherwise

6) `true` if the contents of the `lhs` are lexicographically *greater* than or *equal* the contents of `rhs`, `false` otherwise

Complexity

Linear in the size of the container

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/operator_cmp&oldid=50565"

std::swap(std::map)

```
template< class Key, class T, class Compare, class Alloc >
void swap( map<Key,T,Compare,Alloc>& lhs,
           map<Key,T,Compare,Alloc>& rhs );
```

Specializes the `std::swap` algorithm for `std::map`. Swaps the contents of `lhs` and `rhs`. Calls `lhs.swap(rhs)`.

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Constant.

Exceptions

(since C++17)

`noexcept` specification:

```
noexcept(noexcept(lhs.swap(rhs)))
```

See also

`swap`	swaps the contents (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/map/swap2&oldid=50570"