class template

<map>

**std::map**

template < class Key, // map::key\_type

class T, // map::mapped\_type

class Compare = less<Key>, // map::key\_compare

class Alloc = allocator<pair<const Key,T> > // map::allocator\_type

> class map;

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.  
  
In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value\_type, which is a [pair](http://www.cplusplus.com/pair) type combining both:

|  |  |
| --- | --- |
|  | *typedef* pair<*const* Key, T> value\_type; |

Internally, the elements in a map are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/map::key_comp) (of type Compare).  
  
map containers are generally slower than [unordered\_map](http://www.cplusplus.com/unordered_map) containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.  
  
The mapped values in a [map](http://www.cplusplus.com/map) can be accessed directly by their corresponding key using the *bracket operator* (([operator[]](http://www.cplusplus.com/map::operator%5b%5d)).  
  
Maps are typically implemented as *binary search trees*.

**Container properties**

Associative

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

Map

Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

Unique keys

No two elements in the container can have equivalent *keys*.

Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

**Template parameters**

Key

Type of the *keys*. Each element in a map is uniquely identified by its key value.  
Aliased as member type map::key\_type.

T

Type of the mapped value. Each element in a map stores some data as its mapped value.  
Aliased as member type map::mapped\_type.

Compare

A binary predicate that takes two element keys as arguments and returns a bool. The expression comp(a,b), where *comp* is an object of this type and *a* and *b* are key values, shall return true if *a* is considered to go before *b* in the *strict weak ordering* the function defines.  
The map object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if !comp(a,b) && !comp(b,a)). No two elements in a map container can have equivalent keys.  
This can be a function pointer or a function object (see [constructor](http://www.cplusplus.com/map::map) for an example). This defaults to [less](http://www.cplusplus.com/less)<T>, which returns the same as applying the *less-than operator* (a<b).  
Aliased as member type map::key\_compare.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type map::allocator\_type.

**Member types**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| key\_type | The first template parameter (Key) |  |
| mapped\_type | The second template parameter (T) |  |
| value\_type | [pair](http://www.cplusplus.com/pair)<const key\_type,mapped\_type> |  |
| key\_compare | The third template parameter (Compare) | defaults to: [less](http://www.cplusplus.com/less)<key\_type> |
| value\_compare | *Nested function class to compare elements* | see [value\_comp](http://www.cplusplus.com/map::value_comp) |
| allocator\_type | The fourth template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<value\_type> |
| reference | allocator\_type::reference | for the default [allocator](http://www.cplusplus.com/allocator): value\_type& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type& |
| pointer | allocator\_type::pointer | for the default [allocator](http://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to: iterator\_traits<iterator>::difference\_type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

**Member functions**

[**(constructor)**](http://www.cplusplus.com/reference/map/map/map/)

Construct map (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/map/map/%7Emap/)

Map destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/map/map/operator=/)

Copy container content (public member function )

**Iterators**:

[**begin**](http://www.cplusplus.com/reference/map/map/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/map/map/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/map/map/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/map/map/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/map/map/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/map/map/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/map/map/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/map/map/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**empty**](http://www.cplusplus.com/reference/map/map/empty/)

Test whether container is empty (public member function )

[**size**](http://www.cplusplus.com/reference/map/map/size/)

Return container size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/map/map/max_size/)

Return maximum size (public member function )

**Element access**:

[**operator[]**](http://www.cplusplus.com/reference/map/map/operator%5b%5d/)

Access element (public member function )

[**at**](http://www.cplusplus.com/reference/map/map/at/)

Access element (public member function )

**Modifiers**:

[**insert**](http://www.cplusplus.com/reference/map/map/insert/)

Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/map/map/erase/)

Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/map/map/swap/)

Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/map/map/clear/)

Clear content (public member function )

[**emplace**](http://www.cplusplus.com/reference/map/map/emplace/)

Construct and insert element (public member function )

[**emplace\_hint**](http://www.cplusplus.com/reference/map/map/emplace_hint/)

Construct and insert element with hint (public member function )

**Observers**:

[**key\_comp**](http://www.cplusplus.com/reference/map/map/key_comp/)

Return key comparison object (public member function )

[**value\_comp**](http://www.cplusplus.com/reference/map/map/value_comp/)

Return value comparison object (public member function )

**Operations**:

[**find**](http://www.cplusplus.com/reference/map/map/find/)

Get iterator to element (public member function )

[**count**](http://www.cplusplus.com/reference/map/map/count/)

Count elements with a specific key (public member function )

[**lower\_bound**](http://www.cplusplus.com/reference/map/map/lower_bound/)

Return iterator to lower bound (public member function )

[**upper\_bound**](http://www.cplusplus.com/reference/map/map/upper_bound/)

Return iterator to upper bound (public member function )

[**equal\_range**](http://www.cplusplus.com/reference/map/map/equal_range/)

Get range of equal elements (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/map/map/get_allocator/)

Get allocator (public member function )

You are using a version without Ads of this website. *Please, consider donating:*

Top of Form



Bottom of Form

[[hide](javascript:donate.hide())]

public member function

<map>

**std::[map](http://www.cplusplus.com/reference/map/map/)::map**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **empty (1)** | explicit map (const key\_compare& comp = key\_compare(),  const allocator\_type& alloc = allocator\_type()); |
| **range (2)** | template <class InputIterator>  map (InputIterator first, InputIterator last,  const key\_compare& comp = key\_compare(),  const allocator\_type& alloc = allocator\_type()); |
| **copy (3)** | map (const map& x); |

Construct map

Constructs a [map](http://www.cplusplus.com/map) container object, initializing its contents depending on the constructor version used:

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

(1) empty container constructor (default constructor)

Constructs an [empty](http://www.cplusplus.com/map::empty) container, with no elements.

(2) range constructor

Constructs a container with as many elements as the range [first,last), with each element constructed from its corresponding element in that range.

(3) copy constructor

Constructs a container with a copy of each of the elements in *x*.

The container keeps an internal copy of *alloc* and *comp*, which are used to allocate storage and to sort the elements throughout its lifetime.  
The copy constructor *(3)* creates a container that keeps and uses copies of *x*'s [allocator](http://www.cplusplus.com/map::get_allocator) and [comparison object](http://www.cplusplus.com/map::key_comp).  
  
The storage for the elements is allocated using this [internal allocator](http://www.cplusplus.com/map::get_allocator).

The elements are sorted according to the [comparison object](http://www.cplusplus.com/map::key_comp). If more than one element with equivalent keys is passed to the constructor, only the first one is preserved.

**Parameters**

comp

Binary predicate that, taking two *element keys* as argument, returns true if the first argument goes before the second argument in the *strict weak ordering* it defines, and false otherwise.  
This shall be a function pointer or a function object.  
Member type key\_compare is the internal comparison object type used by the container, defined in [map](http://www.cplusplus.com/map) as an alias of its third template parameter (Compare).  
If key\_compare uses the default [less](http://www.cplusplus.com/less) (which has no state), this parameter is not relevant.

alloc

Allocator object.  
The container keeps and uses an internal copy of this allocator.  
Member type allocator\_type is the internal allocator type used by the container, defined in [map](http://www.cplusplus.com/map) as an alias of its fourth template parameter (Alloc).  
If allocator\_type is an instantiation of the default [allocator](http://www.cplusplus.com/allocator) (which has no state), this parameter is not relevant.

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a range. The range used is [first,last), which includes all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
The function template argument InputIterator shall be an [input iterator](http://www.cplusplus.com/InputIterator) type that points to elements of a type from which value\_type objects can be constructed (in [map](http://www.cplusplus.com/map), value\_type is an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>)

x

Another [map](http://www.cplusplus.com/map) object of the same type (with the same class template arguments Key, T, Compare and Alloc), whose contents are either copied or acquired.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object.  
These objects are automatically constructed from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type> (see [map types](http://www.cplusplus.com/map" \l "types)).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | *// constructing maps*  *#include <iostream>*  *#include <map>*  *bool* fncomp (*char* lhs, *char* rhs) {*return* lhs<rhs;}  *struct* classcomp {  *bool* *operator*() (*const* *char*& lhs, *const* *char*& rhs) *const*  {*return* lhs<rhs;}  };  *int* main ()  {  std::map<*char*,*int*> first;  first['a']=10;  first['b']=30;  first['c']=50;  first['d']=70;  std::map<*char*,*int*> second (first.begin(),first.end());  std::map<*char*,*int*> third (second);  std::map<*char*,*int*,classcomp> fourth; *// class as Compare*  *bool*(\*fn\_pt)(*char*,*char*) = fncomp;  std::map<*char*,*int*,*bool*(\*)(*char*,*char*)> fifth (fn\_pt); *// function pointer as Compare*  *return* 0;  } |

The code does not produce any output, but demonstrates some ways in which a [map](http://www.cplusplus.com/map) container can be constructed.

**Complexity**

Constant for the *empty constructors (1)*, and for the *move constructors (4)* (unless *alloc* is different from *x*'s allocator).  
For all other cases, linear in the distance between the iterators (copy constructions) if the elements are already sorted according to the same criterion. For unsorted sequences, linearithmic (N\*logN) in that distance (sorting,copy constructions).

**Iterator validity**

The *move constructors* (4), invalidate all iterators, pointers and references related to *x* if the elements are moved.

**Data races**

All copied elements are accessed.  
The *move constructors (4)* modify *x*.

**Exception safety**

**Strong guarantee:** no effects in case an exception is thrown.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if the range specified by [first,last) is not valid, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::~map**

~map();

Map destructor

Destroys the container object.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

This destroys all container elements, and deallocates all the storage capacity allocated by the [map](http://www.cplusplus.com/map) container using its [allocator](http://www.cplusplus.com/map::get_allocator).

**Complexity**

Linear in [map::size](http://www.cplusplus.com/map::size) (destructors).

**Iterator validity**

All iterators, pointers and references are invalidated.

**Data races**

The container and all its elements are modified.

**Exception safety**

**No-throw guarantee:** never throws exceptions.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::operator=**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **copy (1)** | map& operator= (const map& x); |

Copy container content

Assigns new contents to the container, replacing its current content.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

Copies all the elements from *x* into the container, changing its [size](http://www.cplusplus.com/map::size) accordingly.  
  
The container preserves its [current allocator](http://www.cplusplus.com/map::get_allocator), which is used to allocate additional storage if needed.

The elements stored in the container before the call are either assigned to or destroyed.

**Parameters**

x

A [map](http://www.cplusplus.com/map) object of the same type (i.e., with the same template parameters, key, T, Compare and Alloc).

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object. The compiler will automatically construct such objects from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type> (see [map member types](http://www.cplusplus.com/map#types)).

**Return value**

\*this

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// assignment operator with maps*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> first;  std::map<*char*,*int*> second;  first['x']=8;  first['y']=16;  first['z']=32;  second=first; *// second now contains 3 ints*  first=std::map<*char*,*int*>(); *// and first is now empty*  std::cout << "Size of first: " << first.size() << '\n';  std::cout << "Size of second: " << second.size() << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| Size of first: 0  Size of second: 3 |

**Complexity**

For the *copy assignment (1)*: Linear in sizes (destructions, copies).  
For the *move assignment (2)*: Linear in current container [size](http://www.cplusplus.com/map::size) (destructions).\*   
For the *initializer list assignment (3)*: Up to logarithmic in sizes (destructions, move-assignments) -- linear if *il* is already sorted.  
\* Additional complexity for assignments if allocators do not [*propagate*](http://www.cplusplus.com/allocator_traits#types).

**Iterator validity**

All iterators, references and pointers related to this container are invalidated.  
  
In the *move assignment*, iterators, pointers and references referring to elements in *x* are also invalidated.

**Data races**

All copied elements are accessed.  
The *move assignment (2)* modifies *x*.  
The container and all its elements are modified.

**Exception safety**

**Basic guarantee:** if an exception is thrown, the container is in a valid state.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if value\_type is not [copy assignable](http://www.cplusplus.com/is_copy_assignable) (or [move assignable](http://www.cplusplus.com/is_move_assignable) for *(2)*), it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::begin**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

iterator begin();

const\_iterator begin() const;

Return iterator to beginning

Returns an iterator referring to the first element in the [map](http://www.cplusplus.com/map) container.  
  
Because [map](http://www.cplusplus.com/map) containers keep their elements ordered at all times, begin points to the element that goes first following the container's [sorting criterion](http://www.cplusplus.com/map::key_comp).  
  
If the container is [empty](http://www.cplusplus.com/map::empty), the returned iterator value shall not be dereferenced.

**Parameters**

none

**Return Value**

An iterator to the first element in the container.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements (of type value\_type).  
Notice that value\_type in [map](http://www.cplusplus.com/map) containers is an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::begin/end*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::iterator it;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  *// show content:*  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| a => 200  b => 100  c => 300 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::end**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

iterator end();

const\_iterator end() const;

Return iterator to end

Returns an iterator referring to the *past-the-end* element in the [map](http://www.cplusplus.com/map) container.  
  
The *past-the-end* element is the theoretical element that would follow the last element in the [map](http://www.cplusplus.com/map) container. It does not point to any element, and thus shall not be dereferenced.  
  
Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [map::begin](http://www.cplusplus.com/map::begin) to specify a range including all the elements in the container.  
  
If the container is [empty](http://www.cplusplus.com/map::empty), this function returns the same as [map::begin](http://www.cplusplus.com/map::begin).

**Parameters**

none

**Return Value**

An iterator to the *past-the-end* element in the container.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// map::begin/end*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  *// show content:*  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| a => 200  b => 100  c => 300 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::rbegin**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

reverse\_iterator rbegin();

const\_reverse\_iterator rbegin() const;

Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the container (i.e., its *reverse beginning*).  
  
*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the container.  
  
rbegin points to the element preceding the one that would be pointed to by member [end](http://www.cplusplus.com/map::end).

**Parameters**

none

**Return Value**

A reverse iterator to the *reverse beginning* of the sequence container.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_reverse\_iterator. Otherwise, it returns a reverse\_iterator.  
  
Member types reverse\_iterator and const\_reverse\_iterator are reverse [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements. See [map member types](http://www.cplusplus.com/map#types).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::rbegin/rend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['x'] = 100;  mymap['y'] = 200;  mymap['z'] = 300;  *// show content:*  std::map<*char*,*int*>::reverse\_iterator rit;  *for* (rit=mymap.rbegin(); rit!=mymap.rend(); ++rit)  std::cout << rit->first << " => " << rit->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| z => 300  y => 200  x => 100 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::rend**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

reverse\_iterator rend();

const\_reverse\_iterator rend() const;

Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element right before the first element in the [map](http://www.cplusplus.com/map) container (which is considered its *reverse end*).  
  
The range between [map::rbegin](http://www.cplusplus.com/map::rbegin) and map::rend contains all the elements of the container (in reverse order).

**Parameters**

none

**Return Value**

A reverse iterator to the *reverse end* of the sequence container.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_reverse\_iterator. Otherwise, it returns a reverse\_iterator.  
  
Member types reverse\_iterator and const\_reverse\_iterator are reverse [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements. See [map member types](http://www.cplusplus.com/map#types).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::rbegin/rend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['x'] = 100;  mymap['y'] = 200;  mymap['z'] = 300;  *// show content:*  std::map<*char*,*int*>::reverse\_iterator rit;  *for* (rit=mymap.rbegin(); rit!=mymap.rend(); ++rit)  std::cout << rit->first << " => " << rit->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| z => 300  y => 200  x => 100 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::cbegin**

const\_iterator cbegin() const noexcept;

Return const\_iterator to beginning

Returns a const\_iterator pointing to the first element in the container.  
  
A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by [map::begin](http://www.cplusplus.com/map::begin), but it cannot be used to modify the contents it points to, even if the [map](http://www.cplusplus.com/map) object is not itself const.  
  
If the container is [empty](http://www.cplusplus.com/map::empty), the returned iterator value shall not be dereferenced.

**Parameters**

none

**Return Value**

A const\_iterator to the beginning of the sequence.  
  
Member type const\_iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to const elements (of type const value\_type).  
Notice that value\_type in [map](http://www.cplusplus.com/map) containers is an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// map::cbegin/cend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  *// print content:*  std::cout << "mymap contains:";  *for* (*auto* it = mymap.cbegin(); it != mymap.cend(); ++it)  std::cout << " [" << (\*it).first << ':' << (\*it).second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains: [a:200] [b:100] [c:300] |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::cend**

const\_iterator cend() const noexcept;

Return const\_iterator to end

Returns a const\_iterator pointing to the *past-the-end* element in the container.  
  
A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by [map::end](http://www.cplusplus.com/map::end), but it cannot be used to modify the contents it points to, even if the [map](http://www.cplusplus.com/map) object is not itself const.  
  
If the container is [empty](http://www.cplusplus.com/map::empty), this function returns the same as [map::cbegin](http://www.cplusplus.com/map::cbegin).  
  
The value returned shall not be dereferenced.

**Parameters**

none

**Return Value**

A const\_iterator to the element past the end of the sequence.  
  
Member type const\_iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to const elements.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// map::cbegin/cend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  *// print content:*  std::cout << "mymap contains:";  *for* (*auto* it = mymap.cbegin(); it != mymap.cend(); ++it)  std::cout << " [" << (\*it).first << ':' << (\*it).second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains: [a:200] [b:100] [c:300] |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::crbegin**

const\_reverse\_iterator crbegin() const noexcept;

Return const\_reverse\_iterator to reverse beginning

Returns a const\_reverse\_iterator pointing to the last element in the container (i.e., its *reverse beginning*).

**Parameters**

none

**Return Value**

A const\_reverse\_iterator to the *reverse beginning* of the sequence.  
  
Member type const\_reverse\_iterator is a reverse [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to a const element (see [map member types](http://www.cplusplus.com/map#types)).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::crbegin/crend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  std::cout << "mymap backwards:";  *for* (*auto* rit = mymap.crbegin(); rit != mymap.crend(); ++rit)  std::cout << " [" << rit->first << ':' << rit->second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap backwards: [c:300] [b:100] [a:200] |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::crend**

const\_reverse\_iterator crend() const noexcept;

Return const\_reverse\_iterator to reverse end

Returns a const\_reverse\_iterator pointing to the theoretical element preceding the first element in the container (which is considered its *reverse end*).

**Parameters**

none

**Return Value**

A const\_reverse\_iterator to the *reverse end* of the sequence.  
  
Member type const\_reverse\_iterator is a reverse [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to a const element (see [map member types](http://www.cplusplus.com/map#types)).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::crbegin/crend*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['b'] = 100;  mymap['a'] = 200;  mymap['c'] = 300;  std::cout << "mymap backwards:";  *for* (*auto* rit = mymap.crbegin(); rit != mymap.crend(); ++rit)  std::cout << " [" << rit->first << ':' << rit->second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap backwards: [c:300] [b:100] [a:200] |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::empty**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

bool empty() const;

Test whether container is empty

Returns whether the [map](http://www.cplusplus.com/map) container is empty (i.e. whether its [size](http://www.cplusplus.com/map::size) is 0).  
  
This function does not modify the container in any way. To clear the content of a [map](http://www.cplusplus.com/map) container, see [map::clear](http://www.cplusplus.com/map::clear).

**Parameters**

none

**Return Value**

true if the container [size](http://www.cplusplus.com/map::size) is 0, false otherwise.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// map::empty*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['a']=10;  mymap['b']=20;  mymap['c']=30;  *while* (!mymap.empty())  {  std::cout << mymap.begin()->first << " => " << mymap.begin()->second << '\n';  mymap.erase(mymap.begin());  }  *return* 0;  } |

Output:

|  |
| --- |
| a => 10  b => 20  c => 30 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::size**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type size() const;

Return container size

Returns the number of elements in the [map](http://www.cplusplus.com/map) container.

**Parameters**

none

**Return Value**

The number of elements in the container.  
  
Member type size\_type is an unsigned integral type.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | *// map::size*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['a']=101;  mymap['b']=202;  mymap['c']=302;  std::cout << "mymap.size() is " << mymap.size() << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap.size() is 3 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::max\_size**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type max\_size() const;

Return maximum size

Returns the maximum number of elements that the [map](http://www.cplusplus.com/map) container can hold.  
  
This is the maximum potential [size](http://www.cplusplus.com/map::size) the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

**Parameters**

none

**Return Value**

The maximum number of elements a [map](http://www.cplusplus.com/map) container can hold as content.  
  
Member type size\_type is an unsigned integral type.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// map::max\_size*  *#include <iostream>*  *#include <map>*  *int* main ()  {  *int* i;  std::map<*int*,*int*> mymap;  *if* (mymap.max\_size()>1000)  {  *for* (i=0; i<1000; i++) mymap[i]=0;  std::cout << "The map contains 1000 elements.\n";  }  *else* std::cout << "The map could not hold 1000 elements.\n";  *return* 0;  } |

Here, member max\_size is used to check beforehand whether the map will allow for 1000 elements to be inserted.

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::operator[]**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

mapped\_type& operator[] (const key\_type& k);

Access element

If *k* matches the key of an element in the container, the function returns a reference to its mapped value.  
  
If *k* does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the [container size](http://www.cplusplus.com/map::size) by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).  
  
A similar member function, [map::at](http://www.cplusplus.com/map::at), has the same behavior when an element with the key exists, but throws an exception when it does not.  
  
A call to this function is equivalent to:  
(\*((*this*->insert(make\_pair(k,mapped\_type()))).first)).second

**Parameters**

k

Key value of the element whose mapped value is accessed.  
Member type key\_type is the type of the keys for the elements stored in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).  
If an rvalue (second version), the key is moved instead of copied when a new element is inserted.

**Return value**

A reference to the mapped value of the element with a key value equivalent to *k*.  
  
Member type mapped\_type is the type of the mapped values in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its second template parameter (T).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// accessing mapped values*  *#include <iostream>*  *#include <map>*  *#include <string>*  *int* main ()  {  std::map<*char*,std::string> mymap;  mymap['a']="an element";  mymap['b']="another element";  mymap['c']=mymap['b'];  std::cout << "mymap['a'] is " << mymap['a'] << '\n';  std::cout << "mymap['b'] is " << mymap['b'] << '\n';  std::cout << "mymap['c'] is " << mymap['c'] << '\n';  std::cout << "mymap['d'] is " << mymap['d'] << '\n';  std::cout << "mymap now contains " << mymap.size() << " elements.\n";  *return* 0;  } |

Notice how the last access (to element 'd') inserts a new element in the [map](http://www.cplusplus.com/map) with that key and initialized to its default value (an empty string) even though it is accessed only to retrieve its value. Member function [map::find](http://www.cplusplus.com/map::find) does not produce this effect.  
Output:

|  |
| --- |
| mymap['a'] is an element  mymap['b'] is another element  mymap['c'] is another element  mymap['d'] is  mymap now contains 4 elements. |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed, and potentially modified.  
The function accesses an element and returns a reference that can be used to modify its mapped value. Concurrently accessing other elements is safe.  
If the function inserts a new element, concurrently iterating ranges in the container is not safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If a new element is inserted and [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) cannot construct an element with *k* and a default-constructed mapped\_type (or if mapped\_type is not [default constructible](http://www.cplusplus.com/is_default_constructible)), it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::at**

mapped\_type& at (const key\_type& k);

const mapped\_type& at (const key\_type& k) const;

Access element

Returns a reference to the mapped value of the element identified with key *k*.  
  
If *k* does not match the key of any element in the container, the function throws an [out\_of\_range](http://www.cplusplus.com/out_of_range) exception.

**Parameters**

k

Key value of the element whose mapped value is accessed.  
Member type key\_type is the type of the keys for the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

A reference to the mapped value of the element with a key value equivalent to *k*.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a reference to const mapped\_type. Otherwise, it returns a reference to mapped\_type.  
  
Member type mapped\_type is the type to the mapped values in the container (see [map member types](http://www.cplusplus.com/map#types)). In [map](http://www.cplusplus.com/map) this is an alias of its second template parameter (T).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// map::at*  *#include <iostream>*  *#include <string>*  *#include <map>*  *int* main ()  {  std::map<std::string,*int*> mymap = {  { "alpha", 0 },  { "beta", 0 },  { "gamma", 0 } };  mymap.at("alpha") = 10;  mymap.at("beta") = 20;  mymap.at("gamma") = 30;  *for* (*auto*& x: mymap) {  std::cout << x.first << ": " << x.second << '\n';  }  *return* 0;  } |

Possible output:

|  |
| --- |
| alpha: 10  beta: 20  gamma: 30 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
The mapped value that is accessed may be modified by the caller. Concurrently accessing or modifying other elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
It throws [out\_of\_range](http://www.cplusplus.com/out_of_range) if *k* is not the key of an element in the [map](http://www.cplusplus.com/map).

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::insert**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **single element (1)** | pair<iterator,bool> insert (const value\_type& val); |
| **with hint (2)** | iterator insert (iterator position, const value\_type& val); |
| **range (3)** | template <class InputIterator>  void insert (InputIterator first, InputIterator last); |

Insert elements

Extends the container by inserting new elements, effectively increasing the container [size](http://www.cplusplus.com/map::size) by the number of elements inserted.  
  
Because element keys in a [map](http://www.cplusplus.com/map) are unique, the insertion operation checks whether each inserted element has a key equivalent to the one of an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).  
  
For a similar container allowing for duplicate elements, see [multimap](http://www.cplusplus.com/multimap).  
  
An alternative way to insert elements in a [map](http://www.cplusplus.com/map) is by using member function [map::operator[]](http://www.cplusplus.com/map::operator%5b%5d).  
  
Internally, [map](http://www.cplusplus.com/map) containers keep all their elements sorted by their key following the criterion specified by its [comparison object](http://www.cplusplus.com/map::key_comp). The elements are always inserted in its respective position following this ordering.  
  
The parameters determine how many elements are inserted and to which values they are initialized:

**Parameters**

val

Value to be copied to (or moved as) the inserted element.  
Member type value\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as [pair](http://www.cplusplus.com/pair)<const key\_type,mapped\_type> (see [map member types](http://www.cplusplus.com/map#types)).  
Template parameter P is a type such that the container can construct an element of type value\_type forwarding *val* as argument.  
If P is instantiated as a reference type, the argument is copied.

position

Hint for the position where the element can be inserted.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

The function optimizes its insertion time if *position* points to the element that will **precede** the inserted element.

Notice that this is just a hint and does not force the new element to be inserted at that position within the [map](http://www.cplusplus.com/map) container (the elements in a [map](http://www.cplusplus.com/map) always follow a specific order depending on their key).  
Member types iterator and const\_iterator are defined in [map](http://www.cplusplus.com/map) as [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types that point to elements.

first, last

Iterators specifying a range of elements. Copies of the elements in the range [first,last) are inserted in the container.  
Notice that the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.  
The function template argument InputIterator shall be an [input iterator](http://www.cplusplus.com/InputIterator) type that points to elements of a type from which value\_type objects can be constructed.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object. Copies of these elements are inserted.  
These objects are automatically constructed from *initializer list* declarators.  
Member type value\_type is the type of the elements contained in the container, defined in [map](http://www.cplusplus.com/map) as [pair](http://www.cplusplus.com/pair)<const key\_type,mapped\_type> (see [map member types](http://www.cplusplus.com/map#types)).

**Return value**

The single element versions (1) return a [pair](http://www.cplusplus.com/pair), with its member pair::first set to an iterator pointing to either the newly inserted element or to the element with an equivalent key in the [map](http://www.cplusplus.com/map). The pair::second element in the [pair](http://www.cplusplus.com/pair) is set to true if a new element was inserted or false if an equivalent key already existed.  
  
The versions with a hint (2) return an iterator pointing to either the newly inserted element or to the element that already had an equivalent key in the [map](http://www.cplusplus.com/map).  
  
Member type iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to elements.  
[pair](http://www.cplusplus.com/pair) is a class template declared in [<utility>](http://www.cplusplus.com/%3Cutility%3E) (see [pair](http://www.cplusplus.com/pair)).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 | *// map::insert (C++98)*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  *// first insert function version (single parameter):*  mymap.insert ( std::pair<*char*,*int*>('a',100) );  mymap.insert ( std::pair<*char*,*int*>('z',200) );  std::pair<std::map<*char*,*int*>::iterator,*bool*> ret;  ret = mymap.insert ( std::pair<*char*,*int*>('z',500) );  *if* (ret.second==*false*) {  std::cout << "element 'z' already existed";  std::cout << " with a value of " << ret.first->second << '\n';  }  *// second insert function version (with hint position):*  std::map<*char*,*int*>::iterator it = mymap.begin();  mymap.insert (it, std::pair<*char*,*int*>('b',300)); *// max efficiency inserting*  mymap.insert (it, std::pair<*char*,*int*>('c',400)); *// no max efficiency inserting*  *// third insert function version (range insertion):*  std::map<*char*,*int*> anothermap;  anothermap.insert(mymap.begin(),mymap.find('c'));  *// showing contents:*  std::cout << "mymap contains:\n";  *for* (it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  std::cout << "anothermap contains:\n";  *for* (it=anothermap.begin(); it!=anothermap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| element 'z' already existed with a value of 200  mymap contains:  a => 100  b => 300  c => 400  z => 200  anothermap contains:  a => 100  b => 300 |

**Complexity**

If a single element is inserted, logarithmic in [size](http://www.cplusplus.com/map::size) in general, but amortized constant if a hint is given and the *position* given is the optimal.

* [C++98](javascript:switch3.select(1))
* [C++11](javascript:switch3.select(2))

If N elements are inserted, Nlog([size](http://www.cplusplus.com/map::size)+N) in general, but linear in [size](http://www.cplusplus.com/map::size)+N if the elements are already sorted according to the same ordering criterion used by the container.

**Iterator validity**

No changes.

**Data races**

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

**Exception safety**

If a single element is to be inserted, there are no changes in the container in case of exception (strong guarantee).  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if an invalid *position* is specified, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::erase**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **(1)** | void erase (iterator position); |
| **(2)** | size\_type erase (const key\_type& k); |
| **(3)** | void erase (iterator first, iterator last); |

Erase elements

Removes from the [map](http://www.cplusplus.com/map) container either a single element or a range of elements ([first,last)).  
  
This effectively reduces the container [size](http://www.cplusplus.com/map::size) by the number of elements removed, which are destroyed.

**Parameters**

position

Iterator pointing to a single element to be removed from the [map](http://www.cplusplus.com/map).  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types that point to elements.

k

Key of the element to be removed from the [map](http://www.cplusplus.com/map).  
Member type key\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

first, last

Iterators specifying a range within the [map](http://www.cplusplus.com/map) container to be removed: [first,last). i.e., the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types that point to elements.

**Return value**

For the key-based version (2), the function returns the number of elements erased, which in [map](http://www.cplusplus.com/map) containers is at most *1*.  
  
Member type size\_type is an unsigned integral type.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

The other versions return no value.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | *// erasing from map*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::iterator it;  *// insert some values:*  mymap['a']=10;  mymap['b']=20;  mymap['c']=30;  mymap['d']=40;  mymap['e']=50;  mymap['f']=60;  it=mymap.find('b');  mymap.erase (it); *// erasing by iterator*  mymap.erase ('c'); *// erasing by key*  it=mymap.find ('e');  mymap.erase ( it, mymap.end() ); *// erasing by range*  *// show content:*  *for* (it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| a => 10  d => 40 |

**Complexity**

For the first version (erase(position)), amortized constant.  
For the second version (erase(val)), logarithmic in container [size](http://www.cplusplus.com/map::size).  
For the last version (erase(first,last)), linear in the distance between *first* and *last*.

**Iterator validity**

Iterators, pointers and references referring to elements removed by the function are invalidated.  
All other iterators, pointers and references keep their validity.

**Data races**

The container is modified.  
The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the container is not.

**Exception safety**

Unless the container's [comparison object](http://www.cplusplus.com/map::key_comp) throws, this function never throws exceptions (no-throw guarantee).  
Otherwise, if a single element is to be removed, there are no changes in the container in case of exception (strong guarantee).  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If an invalid *position* or range is specified, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::swap**

void swap (map& x);

Swap content

Exchanges the content of the container by the content of *x*, which is another [map](http://www.cplusplus.com/map) of the same type. Sizes may differ.  
  
After the call to this member function, the elements in this container are those which were in *x* before the call, and the elements of *x* are those which were in this. All iterators, references and pointers remain valid for the swapped objects.  
  
Notice that a non-member function exists with the same name, [swap](http://www.cplusplus.com/map:swap), overloading that algorithm with an optimization that behaves like this member function.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

Whether the internal container [allocators](http://www.cplusplus.com/map::get_allocator) and [comparison objects](http://www.cplusplus.com/map::key_comp) are swapped is undefined.

**Parameters**

x

Another [map](http://www.cplusplus.com/map) container of the same type as this (i.e., with the same template parameters, Key, T, Compare and Alloc) whose content is swapped with that of this container.

**Return value**

none

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | *// swap maps*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> foo,bar;  foo['x']=100;  foo['y']=200;  bar['a']=11;  bar['b']=22;  bar['c']=33;  foo.swap(bar);  std::cout << "foo contains:\n";  *for* (std::map<*char*,*int*>::iterator it=foo.begin(); it!=foo.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  std::cout << "bar contains:\n";  *for* (std::map<*char*,*int*>::iterator it=bar.begin(); it!=bar.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| foo contains:  a => 11  b => 22  c => 33  bar contains:  x => 100  y => 200 |

**Complexity**

Constant.

**Iterator validity**

All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.  
Note that the *end iterators* do not refer to elements and may be invalidated.

**Data races**

Both the container and *x* are modified.  
No contained elements are accessed by the call (although see *iterator validity* above).

**Exception safety**

If the allocators in both containers compare equal, or if their [allocator traits](http://www.cplusplus.com/allocator_traits) indicate that the allocators shall [propagate](http://www.cplusplus.com/allocator_traits#types), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::clear**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

void clear();

Clear content

Removes all elements from the [map](http://www.cplusplus.com/map) container (which are destroyed), leaving the container with a [size](http://www.cplusplus.com/map::size) of 0.

**Parameters**

none

**Return value**

none

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | *// map::clear*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['x']=100;  mymap['y']=200;  mymap['z']=300;  std::cout << "mymap contains:\n";  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  mymap.clear();  mymap['a']=1101;  mymap['b']=2202;  std::cout << "mymap contains:\n";  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains:  x => 100  y => 200  z => 300  mymap contains:  a => 1101  b => 2202 |

**Complexity**

Linear in [size](http://www.cplusplus.com/map::size) (destructions).

**Iterator validity**

All iterators, pointers and references related to this container are invalidated.

**Data races**

The container is modified.  
All contained elements are modified.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::emplace**

template <class... Args>

pair<iterator,bool> emplace (Args&&... args);

Construct and insert element

Inserts a new element in the [map](http://www.cplusplus.com/map) if its key is unique. This new element is constructed in place using *args* as the arguments for the construction of a value\_type (which is an object of a [pair](http://www.cplusplus.com/pair) type).  
  
The insertion only takes place if no other element in the container has a key equivalent to the one being emplaced (keys in a [map](http://www.cplusplus.com/map) container are unique).  
  
If inserted, this effectively increases the container [size](http://www.cplusplus.com/map::size) by one.  
  
Internally, [map](http://www.cplusplus.com/map) containers keep all their elements sorted by their key following the criterion specified by its [comparison object](http://www.cplusplus.com/map::key_comp). The element is always inserted in its respective position following this ordering.  
  
The element is constructed in-place by calling [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) with *args* forwarded.  
  
A similar member function exists, [insert](http://www.cplusplus.com/map::insert), which either copies or moves existing objects into the container.

**Parameters**

args

Arguments forwarded to construct the new element (of type [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>).

**Return value**

If the function successfully inserts the element (because no equivalent element existed already in the [map](http://www.cplusplus.com/map)), the function returns a [pair](http://www.cplusplus.com/pair) of an iterator to the newly inserted element and a value of true.  
  
Otherwise, it returns an iterator to the equivalent element within the container and a value of false.  
  
Member type iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to an element.  
[pair](http://www.cplusplus.com/pair) is a class template declared in [<utility>](http://www.cplusplus.com/%3Cutility%3E) (see [pair](http://www.cplusplus.com/pair)).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// map::emplace*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap.emplace('x',100);  mymap.emplace('y',200);  mymap.emplace('z',100);  std::cout << "mymap contains:";  *for* (*auto*& x: mymap)  std::cout << " [" << x.first << ':' << x.second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains: [x:100] [y:200] [z:100] |

**Complexity**

Logarithmic in the container [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::emplace\_hint**

template <class... Args>

iterator emplace\_hint (const\_iterator position, Args&&... args);

Construct and insert element with hint

Inserts a new element in the [map](http://www.cplusplus.com/map) if its key is unique, with a hint on the insertion *position*. This new element is constructed in place using *args* as the arguments for the construction of a value\_type (which is an object of a [pair](http://www.cplusplus.com/pair) type).  
  
The insertion only takes place if no other element in the container has a key equivalent to the one being emplaced (elements in a [map](http://www.cplusplus.com/map) container are unique).  
  
If inserted, this effectively increases the container [size](http://www.cplusplus.com/map::size) by one.  
  
The value in *position* is used as a hint on the insertion point. The element will nevertheless be inserted at its corresponding position following the order described by its internal [comparison object](http://www.cplusplus.com/map::key_comp), but this hint is used by the function to begin its search for the insertion point, speeding up the process considerably when the actual insertion point is either *position* or close to it.  
  
The element is constructed in-place by calling [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) with *args* forwarded.

**Parameters**

position

Hint for the position where the element can be inserted.  
The function optimizes its insertion time if *position* points to the element that will follow the inserted element (or to the [end](http://www.cplusplus.com/map::end), if it would be the last).  
Notice that this does not force the new element to be in that position within the [map](http://www.cplusplus.com/map) container (the elements in a [map](http://www.cplusplus.com/map) always follow a specific order).  
const\_iterator is a member type, defined as a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to elements.

args

Arguments forwarded to construct the new element.

**Return value**

If the function successfully inserts the element (because no equivalent element existed already in the [map](http://www.cplusplus.com/map)), the function returns an iterator to the newly inserted element.  
  
Otherwise, it returns an iterator to the equivalent element within the container.  
  
Member type iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to an element.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// map::emplace\_hint*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  *auto* it = mymap.end();  it = mymap.emplace\_hint(it,'b',10);  mymap.emplace\_hint(it,'a',12);  mymap.emplace\_hint(mymap.end(),'c',14);  std::cout << "mymap contains:";  *for* (*auto*& x: mymap)  std::cout << " [" << x.first << ':' << x.second << ']';  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains: [a:12] [b:10] [c:14] |

**Complexity**

Generally, logarithmic in the container [size](http://www.cplusplus.com/map::size).  
Amortized constant if the insertion point for the element is *position*.

**Iterator validity**

No changes.

**Data races**

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments, it causes *undefined behavior*.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::key\_comp**

key\_compare key\_comp() const;

Return key comparison object

Returns a copy of the *comparison object* used by the container to compare *keys*.  
  
The *comparison object* of a [map](http://www.cplusplus.com/map) object is set on [*construction*](http://www.cplusplus.com/map::map). Its type (member key\_compare) is the third template parameter of the [map](http://www.cplusplus.com/map) template. By default, this is a [less](http://www.cplusplus.com/less) object, which returns the same as operator<.  
  
This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the element keys, and returns true if the first argument is considered to go before the second in the *strict weak ordering* it defines, and false otherwise.  
  
Two keys are considered equivalent if key\_comp returns false reflexively (i.e., no matter the order in which the keys are passed as arguments).

**Parameters**

none

**Return value**

The comparison object.  
Member type key\_compare is the type of the *comparison object* associated to the container, defined in [map](http://www.cplusplus.com/map) as an alias of its third template parameter (Compare).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | *// map::key\_comp*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::key\_compare mycomp = mymap.key\_comp();  mymap['a']=100;  mymap['b']=200;  mymap['c']=300;  std::cout << "mymap contains:\n";  *char* highest = mymap.rbegin()->first; *// key value of last element*  std::map<*char*,*int*>::iterator it = mymap.begin();  *do* {  std::cout << it->first << " => " << it->second << '\n';  } *while* ( mycomp((\*it++).first, highest) );  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains:  a => 100  b => 200  c => 300 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::value\_comp**

value\_compare value\_comp() const;

Return value comparison object

Returns a comparison object that can be used to compare two elements to get whether the key of the first one goes before the second.  
  
The arguments taken by this function object are of member type value\_type (defined in [map](http://www.cplusplus.com/map) as an alias of [pair](http://www.cplusplus.com/pair)<const key\_type,mapped\_type>), but the mapped\_type part of the value is not taken into consideration in this comparison.  
  
The comparison object returned is an object of the member type map::value\_compare, which is a nested class that uses the internal [comparison object](http://www.cplusplus.com/map::key_comp) to generate the appropriate comparison functional class. It is defined with the same behavior as:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *template* <*class* Key, *class* T, *class* Compare, *class* Alloc>  *class* map<Key,T,Compare,Alloc>::value\_compare  { *// in C++98, it is required to inherit binary\_function<value\_type,value\_type,bool>*  *friend* *class* map;  *protected*:  Compare comp;  value\_compare (Compare c) : comp(c) {} *// constructed with map's comparison object*  *public*:  *typedef* *bool* result\_type;  *typedef* value\_type first\_argument\_type;  *typedef* value\_type second\_argument\_type;  *bool* *operator*() (*const* value\_type& x, *const* value\_type& y) *const*  {  *return* comp(x.first, y.first);  }  } |

The public member of this comparison class returns true if the key of the first argument is considered to go before that of the second (according to the *strict weak ordering* specified by the container's [comparison object](http://www.cplusplus.com/map::key_comp), [key\_comp](http://www.cplusplus.com/map::key_comp)), and false otherwise.  
  
Notice that value\_compare has no public constructor, therefore no objects can be directly created from this nested class outside [map](http://www.cplusplus.com/map) members.

**Parameters**

none

**Return value**

The comparison object for element values.  
Member type value\_compare is a nested class type (described above).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// map::value\_comp*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['x']=1001;  mymap['y']=2002;  mymap['z']=3003;  std::cout << "mymap contains:\n";  std::pair<*char*,*int*> highest = \*mymap.rbegin(); *// last element*  std::map<*char*,*int*>::iterator it = mymap.begin();  *do* {  std::cout << it->first << " => " << it->second << '\n';  } *while* ( mymap.value\_comp()(\*it++, highest) );  *return* 0;  } |

Output:

|  |
| --- |
| mymap contains:  x => 1001  y => 2002  z => 3003 |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::find**

iterator find (const key\_type& k);

const\_iterator find (const key\_type& k) const;

Get iterator to element

Searches the container for an element with a *key* equivalent to *k* and returns an iterator to it if found, otherwise it returns an iterator to [map::end](http://www.cplusplus.com/map::end).  
  
Two *keys* are considered equivalent if the container's [comparison object](http://www.cplusplus.com/map::key_comp) returns false reflexively (i.e., no matter the order in which the elements are passed as arguments).  
  
Another member function, [map::count](http://www.cplusplus.com/map::count), can be used to just check whether a particular key exists.

**Parameters**

k

Key to be searched for.  
Member type key\_type is the type of the keys for the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

An iterator to the element, if an element with specified key is found, or [map::end](http://www.cplusplus.com/map::end) otherwise.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements (of type value\_type).  
Notice that value\_type in [map](http://www.cplusplus.com/map) containers is an alias of [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | *// map::find*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::iterator it;  mymap['a']=50;  mymap['b']=100;  mymap['c']=150;  mymap['d']=200;  it=mymap.find('b');  mymap.erase (it);  mymap.erase (mymap.find('d'));  *// print content:*  std::cout << "elements in mymap:" << '\n';  std::cout << "a => " << mymap.find('a')->second << '\n';  std::cout << "c => " << mymap.find('c')->second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| elements in mymap:  a => 50  c => 150 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::count**

size\_type count (const key\_type& k) const;

Count elements with a specific key

Searches the container for elements with a key equivalent to *k* and returns the number of matches.  
  
Because all elements in a [map](http://www.cplusplus.com/map) container are unique, the function can only return *1* (if the element is found) or zero (otherwise).  
  
Two *keys* are considered equivalent if the container's [comparison object](http://www.cplusplus.com/map::key_comp) returns false reflexively (i.e., no matter the order in which the keys are passed as arguments).

**Parameters**

k

Key to search for.  
Member type key\_type is the type of the element keys in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

1 if the container contains an element whose key is equivalent to *k*, or zero otherwise.  
  
Member type size\_type is an unsigned integral type.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | *// map::count*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  *char* c;  mymap ['a']=101;  mymap ['c']=202;  mymap ['f']=303;  *for* (c='a'; c<'h'; c++)  {  std::cout << c;  *if* (mymap.count(c)>0)  std::cout << " is an element of mymap.\n";  *else*  std::cout << " is not an element of mymap.\n";  }  *return* 0;  } |

Output:

|  |
| --- |
| a is an element of mymap.  b is not an element of mymap.  c is an element of mymap.  d is not an element of mymap.  e is not an element of mymap.  f is an element of mymap.  g is not an element of mymap. |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::lower\_bound**

iterator lower\_bound (const key\_type& k);

const\_iterator lower\_bound (const key\_type& k) const;

Return iterator to lower bound

Returns an iterator pointing to the first element in the container whose key is not considered to go before *k* (i.e., either it is equivalent or goes after).  
  
The function uses its internal [comparison object](http://www.cplusplus.com/map::key_comp) ([key\_comp](http://www.cplusplus.com/map::key_comp)) to determine this, returning an iterator to the first element for which key\_comp(element\_key,k) would return false.  
  
If the [map](http://www.cplusplus.com/map) class is instantiated with the default comparison type ([less](http://www.cplusplus.com/less)), the function returns an iterator to the first element whose key is not less than *k*.  
  
A similar member function, [upper\_bound](http://www.cplusplus.com/map::upper_bound), has the same behavior as lower\_bound, except in the case that the [map](http://www.cplusplus.com/map) contains an element with a key equivalent to *k*: In this case, lower\_bound returns an iterator pointing to that element, whereas [upper\_bound](http://www.cplusplus.com/map::upper_bound) returns an iterator pointing to the next element.

**Parameters**

k

Key to search for.  
Member type key\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

An iterator to the the first element in the container whose key is not considered to go before *k*, or [map::end](http://www.cplusplus.com/map::end) if all keys are considered to go before *k*.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements (of type value\_type).  
Notice that value\_type in [map](http://www.cplusplus.com/map) containers is itself also a [pair](http://www.cplusplus.com/pair) type: [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | *// map::lower\_bound/upper\_bound*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::iterator itlow,itup;  mymap['a']=20;  mymap['b']=40;  mymap['c']=60;  mymap['d']=80;  mymap['e']=100;  itlow=mymap.lower\_bound ('b'); *// itlow points to b*  itup=mymap.upper\_bound ('d'); *// itup points to e (not d!)*  mymap.erase(itlow,itup); *// erases [itlow,itup)*  *// print content:*  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

|  |
| --- |
| a => 20  e => 100 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::upper\_bound**

iterator upper\_bound (const key\_type& k);

const\_iterator upper\_bound (const key\_type& k) const;

Return iterator to upper bound

Returns an iterator pointing to the first element in the container whose key is considered to go after *k*.  
  
The function uses its internal [comparison object](http://www.cplusplus.com/map::key_comp) ([key\_comp](http://www.cplusplus.com/map::key_comp)) to determine this, returning an iterator to the first element for which key\_comp(k,element\_key) would return true.  
  
If the [map](http://www.cplusplus.com/map) class is instantiated with the default comparison type ([less](http://www.cplusplus.com/less)), the function returns an iterator to the first element whose key is greater than *k*.  
  
A similar member function, [lower\_bound](http://www.cplusplus.com/map::lower_bound), has the same behavior as upper\_bound, except in the case that the [map](http://www.cplusplus.com/map) contains an element with a key equivalent to *k*: In this case [lower\_bound](http://www.cplusplus.com/map::lower_bound) returns an iterator pointing to that element, whereas upper\_bound returns an iterator pointing to the next element.

**Parameters**

k

Key to search for.  
Member type key\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

An iterator to the the first element in the container whose key is considered to go after *k*, or [map::end](http://www.cplusplus.com/map::end) if no keys are considered to go after *k*.  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | *// map::lower\_bound/upper\_bound*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  std::map<*char*,*int*>::iterator itlow,itup;  mymap['a']=20;  mymap['b']=40;  mymap['c']=60;  mymap['d']=80;  mymap['e']=100;  itlow=mymap.lower\_bound ('b'); *// itlow points to b*  itup=mymap.upper\_bound ('d'); *// itup points to e (not d!)*  mymap.erase(itlow,itup); *// erases [itlow,itup)*  *// print content:*  *for* (std::map<*char*,*int*>::iterator it=mymap.begin(); it!=mymap.end(); ++it)  std::cout << it->first << " => " << it->second << '\n';  *return* 0;  } |

|  |
| --- |
| a => 20  e => 100 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::equal\_range**

pair<const\_iterator,const\_iterator> equal\_range (const key\_type& k) const;

pair<iterator,iterator> equal\_range (const key\_type& k);

Get range of equal elements

Returns the bounds of a range that includes all the elements in the container which have a *key* equivalent to *k*.  
  
Because the elements in a [map](http://www.cplusplus.com/map) container have unique keys, the range returned will contain a single element at most.  
  
If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that has a key considered to go after *k* according to the container's [internal comparison object](http://www.cplusplus.com/map::key_comp) ([key\_comp](http://www.cplusplus.com/map::key_comp)).  
  
Two *keys* are considered equivalent if the container's [comparison object](http://www.cplusplus.com/map::key_comp) returns false reflexively (i.e., no matter the order in which the keys are passed as arguments).

**Parameters**

k

Key to search for.  
Member type key\_type is the type of the elements in the container, defined in [map](http://www.cplusplus.com/map) as an alias of its first template parameter (Key).

**Return value**

The function returns a [pair](http://www.cplusplus.com/pair), whose member pair::first is the lower bound of the range (the same as [lower\_bound](http://www.cplusplus.com/map::lower_bound)), and pair::second is the upper bound (the same as [upper\_bound](http://www.cplusplus.com/map::upper_bound)).  
  
If the [map](http://www.cplusplus.com/map) object is const-qualified, the function returns a [pair](http://www.cplusplus.com/pair) of const\_iterator. Otherwise, it returns a [pair](http://www.cplusplus.com/pair) of iterator.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements (of type value\_type).  
Notice that value\_type in [map](http://www.cplusplus.com/map) containers is itself also a [pair](http://www.cplusplus.com/pair) type: [pair](http://www.cplusplus.com/pair)<const key\_type, mapped\_type>.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// map::equal\_range*  *#include <iostream>*  *#include <map>*  *int* main ()  {  std::map<*char*,*int*> mymap;  mymap['a']=10;  mymap['b']=20;  mymap['c']=30;  std::pair<std::map<*char*,*int*>::iterator,std::map<*char*,*int*>::iterator> ret;  ret = mymap.equal\_range('b');  std::cout << "lower bound points to: ";  std::cout << ret.first->first << " => " << ret.first->second << '\n';  std::cout << "upper bound points to: ";  std::cout << ret.second->first << " => " << ret.second->second << '\n';  *return* 0;  } |

|  |
| --- |
| lower bound points to: 'b' => 20  upper bound points to: 'c' => 30 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/map::size).

**Iterator validity**

No changes.

**Data races**

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

**Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

public member function

<map>

**std::**[**map**](http://www.cplusplus.com/reference/map/map/)**::get\_allocator**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

allocator\_type get\_allocator() const;

Get allocator

Returns a copy of the allocator object associated with the [map](http://www.cplusplus.com/map).

**Parameters**

none

**Return Value**

The allocator.  
  
Member type allocator\_type is the type of the allocator used by the container, defined in [map](http://www.cplusplus.com/map) as an alias of its fourth template parameter (Alloc).

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// map::get\_allocator*  *#include <iostream>*  *#include <map>*  *int* main ()  {  *int* psize;  std::map<*char*,*int*> mymap;  std::pair<*const* *char*,*int*>\* p;  *// allocate an array of 5 elements using mymap's allocator:*  p=mymap.get\_allocator().allocate(5);  *// assign some values to array*  psize = *sizeof*(std::map<*char*,*int*>::value\_type)\*5;  std::cout << "The allocated array has a size of " << psize << " bytes.\n";  mymap.get\_allocator().deallocate(p,5);  *return* 0;  } |

The example shows an elaborate way to allocate memory for an array of pairs using the same allocator used by the container.  
A possible output is:

|  |
| --- |
| The allocated array has a size of 40 bytes. |

**Complexity**

Constant.

**Iterator validity**

No changes.

**Data races**

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**

**No-throw guarantee:** this member function never throws exceptions.  
Copying any instantiation of the [default allocator](http://www.cplusplus.com/allocator) is also guaranteed to never throw.