Map in C++ Standard Template Library (STL)

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

|  |  |  |
| --- | --- | --- |
| **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 | value\_type& |  |
| const\_reference | const value\_type& |  |
| pointer | [allocator\_traits](http://www.cplusplus.com/allocator_traits)<allocator\_type>::pointer | for the default [allocator](http://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | [allocator\_traits](http://www.cplusplus.com/allocator_traits)<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/)

***(1) empty container constructors (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 *emplace-constructed* from its corresponding element in that range.

***(3) copy constructor (and copying with allocator)***

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

***(4) move constructor (and moving with allocator)***

Constructs a container that acquires the elements of *x*.  
If *alloc* is specified and is different from *x*'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).  
*x* is left in an unspecified but valid state.

***(5) initializer list constructor***

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

// 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;

}

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

**Copy container content**

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

|  |  |
| --- | --- |
| // 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/operator=/) |

Output:

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

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 )

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

### 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/cbegin/) |

Output:

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

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

**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).

Type of Insertions:

Single Parameter -> Use of pair

Hint Position -> Use of iterators

Range Insertion -> Use of from and to iterator

|  |  |
| --- | --- |
| *single element (1)* | pair<iterator,bool> insert (const value\_type& val);  template <class P> pair<iterator,bool> insert (P&& val); |
| *with hint (2)* | iterator insert (const\_iterator position, const value\_type& val);  template <class P> iterator insert (const\_iterator position, P&& val); |
| *range (3)* | template <class InputIterator>  void insert (InputIterator first, InputIterator last); |
| *initializer list (4)* | void insert (initializer\_list<value\_type> il); |

### Return value

The single element insert 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.

### 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/insert/) |

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 |

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

**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.

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

### Parameters

position

Iterator pointing to a single element to be removed from the [map](http://www.cplusplus.com/map).  
This shall point to a valid and dereferenceable element.  
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.  
  
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 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/erase/) |

Output:

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

# 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.

### 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/swap/) |

Output:

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

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

void clear() noexcept;

**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.

# 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.

### 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::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');  if (it != mymap.end()) // To check if element found.  mymap.erase (it);  // print content:  std::cout << "elements in mymap:" << '\n';  std::cout << "a => " << mymap.find('a')->second << '\n';  std::cout << "c => " << mymap.find('c')->second << '\n';  std::cout << "d => " << mymap.find('d')->second << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/find/) |

Output:

|  |
| --- |
| elements in mymap:  a => 50  c => 150  d => 200 |

**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).

### Return value

1 if the container contains an element whose key is equivalent to *k*, or zero otherwise.

### 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;  } | [Edit & Run](http://www.cplusplus.com/reference/map/map/count/) |

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. |

|  |
| --- |
| / C++ program to illustrate  // the map::count() function  #include <bits/stdc++.h>  using namespace std;    int main()  {        // initialize container      map<int, int> mp;        // insert elements in random order      mp.insert({ 2, 30 });      mp.insert({ 1, 40 });      mp.insert({ 3, 60 });      mp.insert({ 4, 20 });      mp.insert({ 5, 50 });        // checks if key 1 is present or not      if (mp.count(1))          cout << "The key 1 is present\n";      else          cout << "The key 1 is not present\n";        // checks if key 100 is present or not      if (mp.count(100))          cout << "The key 100 is present\n";      else          cout << "The key 100 is not present\n";        return 0;  } |

**Output:**

The key 1 is present

The key 100 is not present

# 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.

// 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['f']=100;

mymap['g']=110;

itlow=mymap.lower\_bound ('a'); // itlow points to b

std::cout <<"Lower bound = " << itlow->first << " => " << itlow->second << '\n';

itlow=mymap.lower\_bound ('c'); // itlow points to d

std::cout <<"Lower bound = " << itlow->first << " => " << itlow->second << '\n';

itlow=mymap.lower\_bound ('b'); // itlow points to b

std::cout <<"Lower bound = " << itlow->first << " => " << itlow->second << '\n';

itup=mymap.upper\_bound ('e'); // itup points to e (not d!)

std::cout<<"Upper bound = " << itup->first << " => " << itup->second << '\n';

itup=mymap.upper\_bound ('f'); // itup points to e (not d!)

std::cout<<"Upper bound = " << itup->first << " => " << itup->second << '\n';

itup=mymap.upper\_bound ('g'); // itup points to e (not d!)

std::cout<<"Upper bound = " << itup->first << " => " << itup->second << '\n';

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;

}

Lower bound = a => 20

Lower bound = d => 80

Lower bound = b => 40

Upper bound = f => 100

Upper bound = g => 110

Upper bound = => 0

a => 20