

# std::multimap

Defined in header <map>

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

Multimap is an associative container that contains a sorted list of key-value pairs. Sorting is done according to the comparison function Compare, applied to the keys. Search, insertion, and removal operations have logarithmic complexity.

The order of the key-value pairs whose keys compare equivalent is the order of insertion and does not change. (since C++11)

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

## Member types

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

## Member classes

**value\_compare** compares objects of type value\_type (class)

## Member functions

(constructor)	constructs the multimap (public member function)
(destructor)	destructs the multimap (public member function)
<b>operator=</b>	assigns values to the container (public member function)
<b>get_allocator</b>	returns the associated allocator (public member function)

### Iterators

<b>begin</b> <b>cbegin</b>	returns an iterator to the beginning (public member function)
<b>end</b> <b>cend</b>	returns an iterator to the end (public member function)
<b>rbegin</b> <b>crbegin</b>	returns a reverse iterator to the beginning (public member function)
<b>rend</b> <b>crend</b>	returns a reverse iterator to the end (public member function)

## Capacity

<b>empty</b>	checks whether the container is empty (public member function)
<b>size</b>	returns the number of elements (public member function)
<b>max_size</b>	returns the maximum possible number of elements (public member function)

## Modifiers

<b>clear</b>	clears the contents (public member function)
<b>insert</b>	inserts elements (public member function)
<b>emplace</b> (C++11)	constructs element in-place (public member function)
<b>emplace_hint</b> (C++11)	constructs elements in-place using a hint (public member function)
<b>erase</b>	erases elements (public member function)
<b>swap</b>	swaps the contents (public member function)

## Lookup

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

## Observers

<b>key_comp</b>	returns the function that compares keys (public member function)
<b>value_comp</b>	returns the function that compares keys in objects of type value_type (public member function)

## Non-member functions

<b>operator==</b> <b>operator!=</b> <b>operator&lt;</b> <b>operator&lt;=</b> <b>operator&gt;</b> <b>operator&gt;=</b>	lexicographically compares the values in the multimap (function template)
<b>std::swap</b> (std::multimap)	specializes the std::swap algorithm (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap&oldid=57285"

# std::multimap::multimap

<code>explicit multimap( const Compare&amp; comp = Compare(), const Allocator&amp; alloc = Allocator() );</code>	(until C++14)
<code>multimap() : multimap( Compare() ) {}</code>	(1)
<code>explicit multimap( const Compare&amp; comp, const Allocator&amp; alloc = Allocator() );</code>	(since C++14)
<code>explicit multimap( const Allocator&amp; alloc );</code>	(1) (since C++11)
<code>template&lt; class InputIterator &gt; multimap( InputIterator first, InputIterator last, const Compare&amp; comp = Compare(), const Allocator&amp; alloc = Allocator() );</code>	(2)
<code>template&lt; class InputIterator &gt; multimap( InputIterator first, InputIterator last, const Allocator&amp; alloc );</code>	(since C++14)
<code>multimap( const multimap&amp; other );</code>	(3)
<code>multimap( const multimap&amp; other, const Allocator&amp; alloc );</code>	(3) (since C++11)
<code>multimap( multimap&amp;&amp; other );</code>	(4) (since C++11)
<code>multimap( multimap&amp;&amp; other, const Allocator&amp; alloc );</code>	(4) (since C++11)
<code>multimap( std::initializer_list&lt;value_type&gt; init, const Compare&amp; comp = Compare(), const Allocator&amp; alloc = Allocator() );</code>	(5) (since C++11)
<code>multimap( std::initializer_list&lt;value_type&gt; init, const Allocator&amp; );</code>	(since C++14)

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

- 1) Default constructor. Constructs empty container.
- 2) Constructs the container with the contents of the range `[first, last)`.
- 3) Copy constructor. Constructs the container with the copy of the contents of `other`. If `alloc` is not provided, allocator is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.
- 4) Move constructor. Constructs the container with the contents of `other` using move semantics. If `alloc` is not provided, allocator is obtained by move-construction from the allocator belonging to `other`.
- 5) Constructs the container with the contents of the initializer list `init`.

## Parameters

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

## Type requirements

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

## Complexity

- 1) Constant
- 2)  $N \log(N)$  where `N = std::distance(first, last)` in general, linear in `N` if the range is already sorted by `value_comp()`.
- 3) Linear in size of `other`
- 4) Constant. If `alloc` is given and `alloc != other.get_allocator()`, then linear.

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

## Example

This section is incomplete  
Reason: no example

## See also

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

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/multimap&oldid=50589"

## std::multimap::~~multimap

---

```
~multimap();
```

---

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

### Complexity

Linear in the size of the container.

---

Retrieved from "<http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/%7Emultimap&oldid=50599>"

## std::multimap::operator=

<code>multimap&amp; operator=( const multimap&amp; other );</code>	(1)
<code>multimap&amp; operator=( multimap&amp;&amp; other );</code>	(2) (since C++11)
<code>multimap&amp; operator=( std::initializer_list&lt;value_type&gt; ilist );</code>	(3) (since C++11)

Replaces the contents of the container.

- 1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. If `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If the target and the source allocators do not compare equal, the target ( `*this` ) allocator is used to deallocate the memory, then `other`'s allocator is used to allocate it before copying the elements. (since C++11)
- 2) Move assignment operator. Replaces the contents with those of `other` using move semantics (i.e. the data in `other` is moved from `other` into this container). `other` is in a valid but unspecified state afterwards. If `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If it is `false` and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must move-assign each element individually, allocating additional memory using its own allocator as needed.
- 3) Replaces the contents with those identified by initializer list `ilist`.

### Parameters

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

### Return value

`*this`

### Complexity

- 1) Linear in the size of the `other`.
- 2) Constant unless `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is false and the allocators do not compare equal (in which case linear).
- 3) Linear in the size of `ilist`.

### Exceptions

- 2) `noexcept` specification: (since C++17)  
`noexcept(std::allocator_traits<Allocator>::is_always_equal::value  
&& std::is_nothrow_move_assignable<Compare>::value)`

### Example

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

Run this code

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

void display_sizes(const std::multimap<int, int> &nums1,
                  const std::multimap<int, int> &nums2,
```

```

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

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

    std::multimap<int, int> nums2;
    std::multimap<int, int> nums3;

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

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

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

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

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

```

Output:

```

Initially:
nums1: 6 nums2: 0 nums3: 0
After assignment:
nums1: 6 nums2: 6 nums3: 0
After move assignment:
nums1: 0 nums2: 6 nums3: 6

```

## See also

(constructor) constructs the multimap  
 (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/operator%3D&oldid=41915"

## std::multimap::get\_allocator

---

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

---

Returns the allocator associated with the container.

### Parameters

(none)

### Return value

The associated allocator.

### Complexity

Constant.

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/get\\_allocator&oldid=50585](http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/get_allocator&oldid=50585)"

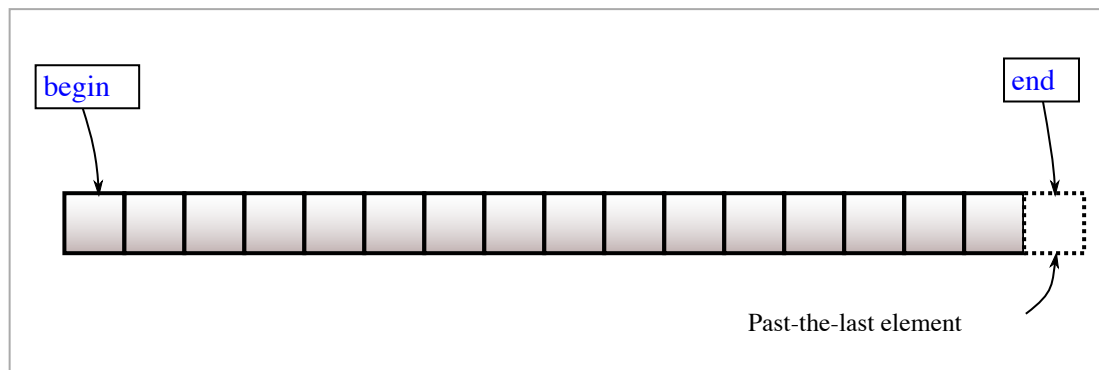


## std::multimap::begin, std::multimap::cbegin

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

Returns an iterator to the first element of the container.

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



### Parameters

(none)

### Return value

Iterator to the first element

### Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

### Complexity

Constant

### Example

This section is incomplete  
Reason: no example

### See also

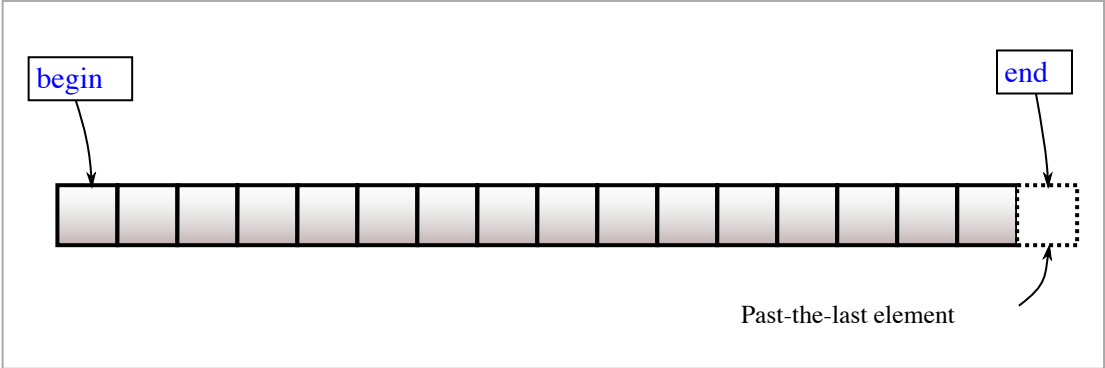
**end** returns an iterator to the end  
**cend** (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/begin&oldid=50575"

# std::multimap::end, std::multimap::cend

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

Returns an iterator to the element following the last element of the container.  
This element acts as a placeholder; attempting to access it results in undefined behavior.



## Parameters

(none)

## Return value

Iterator to the element following the last element.

## Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

## Complexity

Constant.

## See also

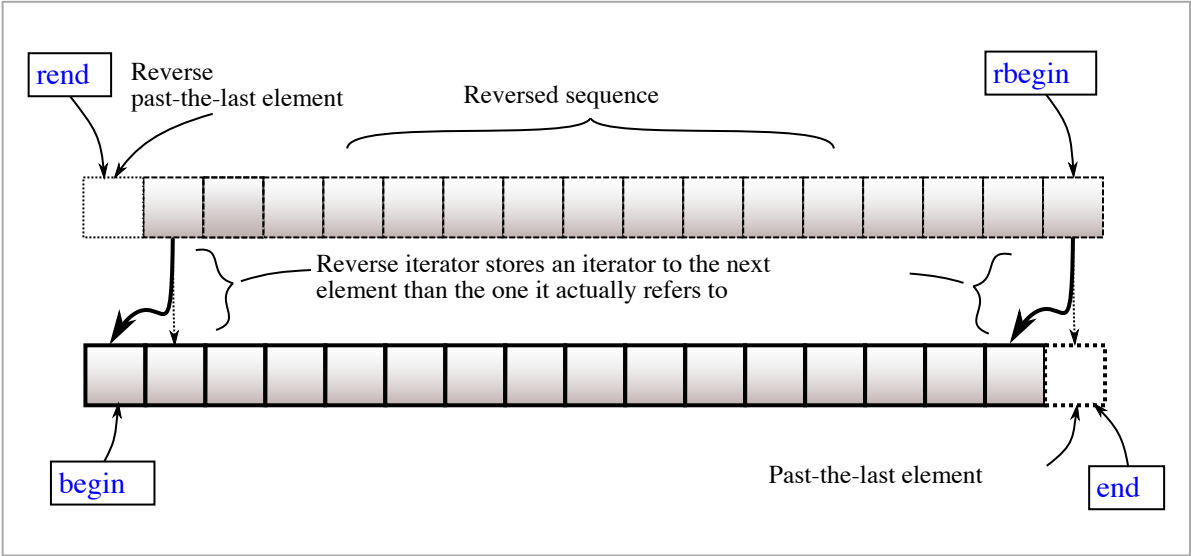
<b>begin</b>	returns an iterator to the beginning
<b>cbegin</b>	(public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/end&oldid=50581"

# std::multimap::rbegin, std::multimap::crbegin

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

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



## Parameters

(none)

## Return value

Reverse iterator to the first element.

## Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

## Complexity

Constant.

## See also

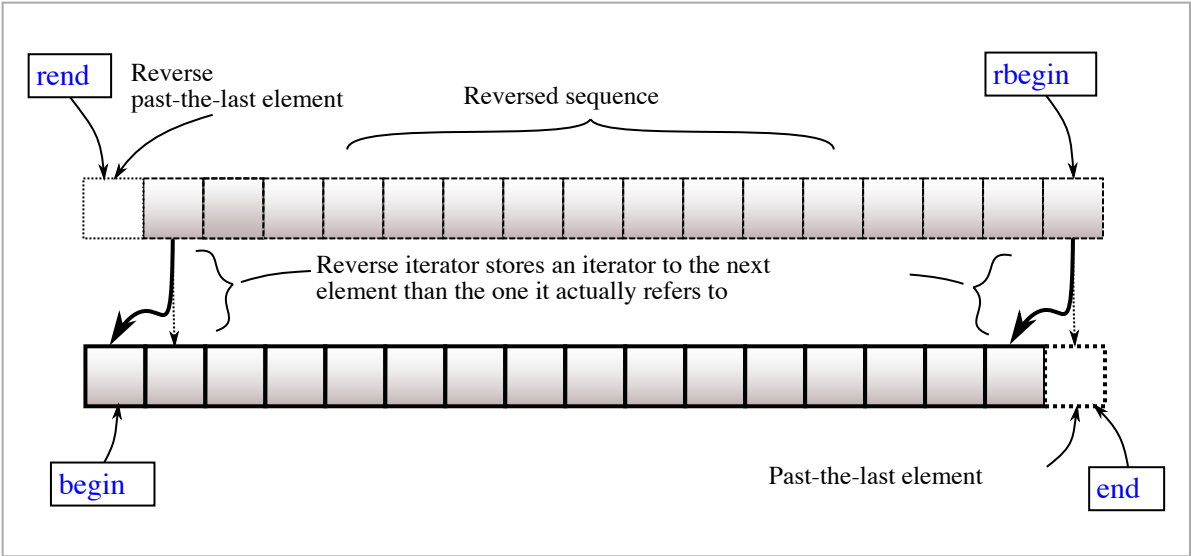
<code>rend</code>	returns a reverse iterator to the end
<code>crend</code>	(public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/rbegin&oldid=50591"

# std::multimap::rend, std::multimap::crend

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

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



## Parameters

(none)

## Return value

Reverse iterator to the element following the last element.

## Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

## Complexity

Constant.

## See also

<code>rbegin</code>	returns a reverse iterator to the beginning
<code>crbegin</code>	(public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/rend&oldid=50592"

## std::multimap::erase

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

Removes specified elements from the container.

- 1) Removes the element at `pos`.
- 2) Removes the elements in the range `[first, last)`, which must be a valid range in `*this`.
- 3) Removes all elements with the key equivalent to `key`.

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

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

### Parameters

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

### Return value

- 1-2) Iterator following the last removed element.
- 3) Number of elements removed.

### Exceptions

- 1,2) (none)
- 3) Any exceptions thrown by the Compare object.

### Complexity

Given an instance `c` of `multimap`:

- 1) Amortized constant
- 2) `log(c.size()) + std::distance(first, last)`
- 3) `log(c.size()) + c.count(k)`

### Example

Run this code

```
#include <map>
#include <iostream>
int main()
{
    std::multimap<int, std::string> c = {{1, "one"}, {2, "two"}, {3, "three"},
                                         {4, "four"}, {5, "five"}, {6, "six"}};
    // erase all odd numbers from c
```

```
for(auto it = c.begin(); it != c.end(); )
    if(it->first % 2 == 1)
        it = c.erase(it);
    else
        ++it;
for(auto& p : c)
    std::cout << p.second << ' ';
}
```

Output:

```
two four six
```

## See also

**clear** clears the contents  
(public member function)

---

Retrieved from "<http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/erase&oldid=50583>"

## std::multimap::size

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

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

### Parameters

(none)

### Return value

The number of elements in the container.

### Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

### Complexity

Constant.

### Example

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

Run this code

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

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

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

Output:

```
nums contains 4 elements.
```

### See also

<b>empty</b>	checks whether the container is empty (public member function)
<b>max_size</b>	returns the maximum possible number of elements (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/size&oldid=50593"

## std::multimap::max\_size

---

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

---

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

### Parameters

(none)

### Return value

Maximum number of elements.

### Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

### Complexity

Constant.

### Notes

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

### Example

Run this code

```
#include <iostream>
#include <multimap>

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

Possible output:

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

### See also

---

**size** returns the number of elements  
(public member function)

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/max\_size&oldid=50588"



## std::multimap::clear

---

```
void clear();
```

---

Removes all elements from the container.

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

### Parameters

(none)

### Return value

(none)

### Exceptions

(none)	(until C++11)
noexcept specification:	<u>noexcept</u> (since C++11)

### Complexity

Linear in the size of the container.

### See also

---

<b>erase</b>	erases elements (public member function)
--------------	---

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/clear&oldid=50576"

## std::multimap::insert

iterator insert( const value_type& value );	(1)	
template< class P > iterator insert( P&& value );	(2)	(since C++11)
iterator insert( value_type&& value );	(2)	(since C++17)
iterator insert( iterator hint, const value_type& value );	(3)	(until C++11)
iterator insert( const_iterator hint, const value_type& value );		(since C++11)
template< class P > iterator insert( const_iterator hint, P&& value );	(4)	(since C++11)
iterator insert( const_iterator hint, value_type&& value );	(4)	(since C++17)
template< class InputIt > void insert( InputIt first, InputIt last );	(5)	
void insert( std::initializer_list<value_type> ilist );	(6)	(since C++11)

Inserts element(s) into the container.

1-2) inserts value. If the container has elements with equivalent key, inserts at the upper bound of that range.(since C++11) The overload (2) is equivalent to `emplace(std::forward<P>(value))` and only participates in overload resolution if

`std::is_constructible<value_type, P&&>::value == true`.

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

5) inserts elements from range [first, last).

6) inserts elements from initializer list ilist.

No iterators or references are invalidated.

### Parameters

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

### Type requirements

- InputIt must meet the requirements of InputIterator.

### Return value

1-4) Returns an iterator to the inserted element.

5-6) (none)

### Exceptions

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

This section is incomplete  
Reason: cases 5-6

### Complexity

1-2) Logarithmic in the size of the container,  $O(\log(\text{size}()))$ .

3-4) Amortized constant if the insertion happens in the position just *after* the hint, logarithmic

in the size of the container otherwise.

(until C++11)

3-4) Amortized constant if the insertion happens in the position just *before* the hint, logarithmic in the size of the container otherwise.

(since C++11)

5-6)  $O(N \cdot \log(\text{size}() + N))$ , where N is the number of elements to insert.

## See also

<b>emplace</b> (C++11)	constructs element in-place (public member function)
<b>emplace_hint</b> (C++11)	constructs elements in-place using a hint (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/insert&oldid=74441"

# std::multimap::emplace

```
template< class... Args >
iterator emplace( Args&&... args );           (since C++11)
```

Inserts a new element into the container by constructing it in-place with the given args .

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

No iterators or references are invalidated.

## Parameters

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

## Return value

Returns an iterator to the inserted element.

## Exceptions

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

## Complexity

Logarithmic in the size of the container.

## Example

Run this code

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

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

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

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

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

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

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

Output:

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

## See also

---

<b>emplace_hint</b> (C++11)	constructs elements in-place using a hint (public member function)
<b>insert</b>	inserts elements (public member function)

---

Retrieved from "<http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/emplace&oldid=50578>"

# std::multimap::emplace\_hint

---

```
template <class... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );           (since C++11)
```

---

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

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

No iterators or references are invalidated.

## Parameters

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

## Return value

Returns an iterator to the newly inserted element.

## Complexity

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

## See also

---

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

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/emplace\_hint&oldid=73439"

## std::multimap::swap

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

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

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

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

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

### Parameters

**other** - container to exchange the contents with

### Return value

(none)

### Exceptions

Any exception thrown by the swap of the Compare objects.

(until C++17)

`noexcept` specification:

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value
&& noexcept(std::swap(std::declval<Compare>(), std::declval<Compare>())))
```

(since C++17)

### Complexity

Constant.

### See also

**std::swap**(std::multimap) specializes the `std::swap` algorithm  
(function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/swap&oldid=50594"

## std::multimap::count

---

```
size_type count( const Key& key ) const;      (1)
template< class K >
size_type count( const K& x ) const;          (2) (since C++14)
```

---

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

### Parameters

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

### Return value

Number of elements with key `key`.

### Complexity

Logarithmic in the size of the container plus linear in the number of the elements found.

### See also

---

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

---

Retrieved from "<http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/count&oldid=65125>"



## std::multimap::find

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

- 1,2) Finds an element with key equivalent to `key`. If there are several elements with `key` in the container, any of them may be returned.
- 3,4) Finds an element with key that compares *equivalent* to the value `x`. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. It allows calling this function without constructing an instance of `Key`

### Parameters

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

### Return value

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

### Complexity

Logarithmic in the size of the container.

### Example

Run this code

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

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

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

Output:

```
Found 2 b
```

### See also

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

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/find&oldid=50584"



## std::multimap::equal\_range

<code>std::pair&lt;iterator,iterator&gt; equal_range( const Key&amp; key );</code>	(1)	
<code>std::pair&lt;const_iterator,const_iterator&gt; equal_range( const Key&amp; key ) const;</code>	(2)	
<code>template&lt; class K &gt;</code> <code>std::pair&lt;iterator,iterator&gt; equal_range( const K&amp; x );</code>	(3)	(since C++14)
<code>template&lt; class K &gt;</code> <code>std::pair&lt;const_iterator,const_iterator&gt; equal_range( const K&amp; x ) const;</code>	(4)	(since C++14)

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

1,2) Compares the keys to key.

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

This section is incomplete  
Reason: explain better

### Parameters

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

### Return value

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

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

Since insert always inserts at the upper bound, the order of equivalent elements in the equal range is the order of insertion. (since C++11)

### Complexity

Logarithmic in the size of the container.

### Example

This section is incomplete  
Reason: no example

### See also

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

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/equal\_range&oldid=65092"

# std::multimap::lower\_bound

<code>iterator lower_bound( const Key&amp; key );</code>	(1)
<code>const_iterator lower_bound( const Key&amp; key ) const;</code>	(1)
<code>template&lt; class K &gt;</code>	
<code>iterator lower_bound(const K&amp; x);</code>	(2) (since C++14)
<code>template&lt; class K &gt;</code>	
<code>const_iterator lower_bound(const K&amp; x) const;</code>	(2) (since C++14)

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

## Parameters

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

## Return value

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

## Complexity

Logarithmic in the size of the container.

## See also

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

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/lower\_bound&oldid=50587"

## std::multimap::upper\_bound

---

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

---

(1)

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

---

(1)

```
template< class K >
```

```
iterator upper_bound( const K& x );
```

---

(2) (since C++14)

```
template< class K >
```

```
const_iterator upper_bound( const K& x ) const;
```

---

(2) (since C++14)

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

### Parameters

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

### Return value

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

### Complexity

Logarithmic in the size of the container.

### See also

---

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

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/upper\_bound&oldid=50596"

## std::multimap::key\_comp

---

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

---

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

### Parameters

(none)

### Return value

The key comparison function object.

### Complexity

Constant.

### See also

---

<b>value_comp</b>	returns the function that compares keys in objects of type <code>value_type</code> (public member function)
-------------------	--

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/key\_comp&oldid=50586"

## std::multimap::value\_comp

---

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

---

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

### Parameters

(none)

### Return value

The value comparison function object.

### Complexity

Constant.

### See also

---

<b>key_comp</b>	returns the function that compares keys (public member function)
-----------------	---

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/value\\_comp&oldid=50597](http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/value_comp&oldid=50597)"

# std::multimap::value\_compare

```
class value_compare;
```

std::multimap::value\_compare is a function object that compares objects of type std::multimap::value\_type (key-value pairs) by comparing of the first components of the pairs.

## Member types

Type	Definition
result_type	<a href="#">bool</a>
first_argument_type	value_type
second_argument_type	value_type

## Protected member objects

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

## Member functions

(constructor) constructs a new value\_compare object  
(protected member function)

**operator()** compares two values of type value\_type  
(public member function)

### std::multimap<Key,T,Compare,Alloc>::value\_compare::value\_compare

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

Initializes the internal instance of the comparator to c.

#### Parameters

**c** - comparator to assign

### std::multimap<Key,T,Compare,Alloc>::value\_compare::operator()

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

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

#### Parameters

**lhs, rhs** - values to compare

#### Return value

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

#### Exceptions



(none)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/value\_compare&oldid=50598"

# std::multimap::empty

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

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

## Parameters

(none)

## Return value

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

## Exceptions

(none)	(until C++11)
noexcept specification:	<code>noexcept</code> (since C++11)

## Complexity

Constant.

## Example

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

Run this code

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

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

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

Output:

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

## See also

**size** returns the number of elements  
(public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/empty&oldid=50580"

## operator==,!=,<,<=,>,>=(std::multimap)

---

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

---

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

---

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

---

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

---

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

---

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

---

Compares the contents of two containers.

- 1-2) Checks if the contents of lhs and rhs are equal, that is, whether `lhs.size() == rhs.size()` and each element in lhs compares equal with the element in rhs at the same position.
- 3-6) Compares the contents of lhs and rhs lexicographically. The comparison is performed by a function equivalent to `std::lexicographical_compare`.

### Parameters

**lhs, rhs** - containers whose contents to compare

- **T**, **Key** must meet the requirements of `EqualityComparable` in order to use overloads (1-2).
- **Key** must meet the requirements of `LessThanComparable` in order to use overloads (3-6). The ordering relation must establish total order.

### Return value

- 1) `true` if the contents of the containers are equal, `false` otherwise
- 2) `true` if the contents of the containers are not equal, `false` otherwise
- 3) `true` if the contents of the lhs are lexicographically *less* than the contents of rhs, `false` otherwise
- 4) `true` if the contents of the lhs are lexicographically *less* than or *equal* the contents of rhs, `false` otherwise
- 5) `true` if the contents of the lhs are lexicographically *greater* than the contents of rhs, `false` otherwise
- 6) `true` if the contents of the lhs are lexicographically *greater* than or *equal* the contents of rhs, `false` otherwise

### Complexity

Linear in the size of the container

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/operator\\_cmp&oldid=50590](http://en.cppreference.com/mwiki/index.php?title=cpp/container/multimap/operator_cmp&oldid=50590)"

## std::swap(std::multimap)

---

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

---

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

### Parameters

**lhs**, **rhs** - containers whose contents to swap

### Return value

(none)

### Complexity

Constant.

### Exceptions

noexcept specification:

(since C++17)

---

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

---

### See also

---

**swap** swaps the contents  
(public member function)

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/multimap/swap2&oldid=50595"