

# std::unordered\_multimap

Defined in header `<unordered_map>`

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_multimap;                                     (since C++11)
```

Unordered multimap is an unordered associative container that supports equivalent keys (an unordered\_multimap may contain multiple copies of each key value) and that associates values of another type with the keys. The unordered\_multimap class supports forward iterators. Search, insertion, and removal have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. This allows fast access to individual elements, since once hash is computed, it refers to the exact bucket the element is placed into.

The iteration order of this container is not required to be stable (so, for example, std::equal cannot be used to compare two std::unordered\_multimaps), except that every group of elements whose keys compare *equivalent* (compare equal with key\_eq() as the comparator) forms a contiguous subrange in the iteration order, also accessible with equal\_range().

std::unordered\_multimap meets the requirements of Container, AllocatorAwareContainer, UnorderedAssociativeContainer.

## 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)
hasher	Hash
key_equal	KeyEqual
allocator_type	Allocator
reference	value_type&
const_reference	const value_type&
pointer	std::allocator_traits<Allocator>::pointer
const_pointer	std::allocator_traits<Allocator>::const_pointer
iterator	ForwardIterator
const_iterator	Constant forward iterator
local_iterator	An iterator type whose category, value, difference, pointer and reference types are the same as iterator. This iterator can be used to iterate through a single bucket but not across buckets
const_local_iterator	An iterator type whose category, value, difference, pointer and reference types are the same as const_iterator. This iterator can be used to iterate through a single bucket but not across buckets

## Member functions

(constructor)	constructs the unordered_multimap (public member function)
(destructor)	destructs the unordered_multimap (public member function)
operator=	assigns values to the container (public member function)
get_allocator	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)

**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>	constructs element in-place (public member function)
<b>emplace_hint</b>	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)

**Bucket interface**

<b>begin</b> (int) <b>cbegin</b> (int)	returns an iterator to the beginning of the specified bucket (public member function)
<b>end</b> (int) <b>cend</b> (int)	returns an iterator to the end of the specified bucket (public member function)
<b>bucket_count</b>	returns the number of buckets (public member function)
<b>max_bucket_count</b>	returns the maximum number of buckets (public member function)
<b>bucket_size</b>	returns the number of elements in specific bucket (public member function)
<b>bucket</b>	returns the bucket for specific key (public member function)

**Hash policy**

<b>load_factor</b>	returns average number of elements per bucket (public member function)
<b>max_load_factor</b>	manages maximum average number of elements per bucket (public member function)
<b>rehash</b>	reserves at least the specified number of buckets. This regenerates the hash table. (public member function)
<b>reserve</b>	reserves space for at least the specified number of elements. This regenerates the hash table. (public member function)

Observers

<b>hash_function</b>	returns function used to hash the keys (public member function)
<b>key_eq</b>	returns the function used to compare keys for equality (public member function)

Non-member functions

<b>operator==</b> <b>operator!=</b>	compares the values in the unordered_multimap (function template)
<b>std::swap</b> (std::unordered_multimap) (C++11)	specializes the std::swap algorithm (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap&oldid=78564"

## std::unordered\_multimap::unordered\_multimap

<code>explicit unordered_multimap( size_type bucket_count = <i>/*implementation-defined*/</i>, const Hash&amp; hash = Hash(), const KeyEqual&amp; equal = KeyEqual(), const Allocator&amp; alloc = Allocator() );</code>	(since C++11) (until C++14)
<code>unordered_multimap() : unordered_multimap( size_type(<i>/*implementation-defined*/</i>) {} explicit unordered_multimap( size_type bucket_count, const Hash&amp; hash = Hash(), const KeyEqual&amp; equal = KeyEqual(), const Allocator&amp; alloc = Allocator() );</code>	(1) (since C++14)
<code>unordered_multimap( size_type bucket_count, const Allocator&amp; alloc ) : unordered_multimap(bucket_count, Hash(), KeyEqual(), alloc) {} unordered_multimap( size_type bucket_count, const Hash&amp; hash, const Allocator&amp; alloc ) : unordered_multimap(bucket_count, hash, KeyEqual(), alloc) {}</code>	(1) (since C++14)
<code>explicit unordered_multimap( const Allocator&amp; alloc );</code>	(1) (since C++11)
<code>template&lt; class InputIt &gt; unordered_multimap( InputIt first, InputIt last, size_type bucket_count = <i>/*implementation-defined*/</i>, const Hash&amp; hash = Hash(), const KeyEqual&amp; equal = KeyEqual(), const Allocator&amp; alloc = Allocator() );</code>	(2) (since C++11)
<code>template&lt; class InputIt &gt; unordered_multimap( InputIt first, InputIt last, size_type bucket_count, const Allocator&amp; alloc ) : unordered_multimap(first, last, bucket_count, Hash(), KeyEqual(), alloc) {}</code>	(2) (since C++14)
<code>template&lt; class InputIt &gt; unordered_multimap( InputIt first, InputIt last, size_type bucket_count, const Hash&amp; hash, const Allocator&amp; alloc ) : unordered_multimap(first, last, bucket_count, hash, KeyEqual(), alloc) {}</code>	(2) (since C++14)
<code>unordered_multimap( const unordered_multimap&amp; other );</code>	(3) (since C++11)
<code>unordered_multimap( const unordered_multimap&amp; other, const Allocator&amp; alloc );</code>	(3) (since C++11)
<code>unordered_multimap( unordered_multimap&amp;&amp; other );</code>	(4) (since C++11)
<code>unordered_multimap( unordered_multimap&amp;&amp; other, const Allocator&amp; alloc );</code>	(4) (since C++11)
<code>unordered_multimap( std::initializer_list&lt;value_type&gt; init, size_type bucket_count = <i>/*implementation-defined*/</i>, const Hash&amp; hash = Hash(), const KeyEqual&amp; equal = KeyEqual(), const Allocator&amp; alloc = Allocator() );</code>	(5) (since C++11)
<code>unordered_multimap( std::initializer_list&lt;value_type&gt; init, size_type bucket_count, const Allocator&amp; alloc ) : unordered_multimap(init, bucket_count, Hash(), KeyEqual(), alloc) {}</code>	(5) (since C++14)
<code>unordered_multimap( std::initializer_list&lt;value_type&gt; init, size_type bucket_count, const Hash&amp; hash, const Allocator&amp; alloc ) : unordered_multimap(init, bucket_count, hash, KeyEqual(), alloc) {}</code>	(5) (since C++14)

Constructs new container from a variety of data sources. Optionally uses user supplied `bucket_count` as a minimal number of buckets to create, `hash` as the hash function, `equal` as the function to compare keys and `alloc` as the allocator.

1) Constructs empty container. Sets `max_load_factor()` to 1.0. For the default constructor, the

number of buckets is implementation-defined.

2) constructs the container with the contents of the range `[first, last)`. Sets `max_load_factor()` to 1.0.

3) copy constructor. Constructs the container with the copy of the contents of `other`, copies the load factor, the predicate, and the hash function as well. 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`, same as `unordered_multimap(init.begin(), init.end())`.

## Parameters

- alloc** - allocator to use for all memory allocations of this container
- bucket\_count** - minimal number of buckets to use on initialization. If it is not specified, implementation-defined default value is used
- hash** - hash function to use
- equal** - comparison function to use for all key comparisons of this container
- 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

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

## Complexity

- 1) constant
- 2) average case linear worst case quadratic in distance between `first` and `last`
- 3) linear in size of `other`
- 4) constant. If `alloc` is given and `alloc != other.get_allocator()`, then linear.
- 5) average case linear worst case quadratic in size of `init`

## 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/unordered\_multimap/unordered\_multimap&oldid=50738"

## std::unordered\_multimap::~~unordered\_multimap

---

```
~unordered_multimap( ); (since C++11)
```

---

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

### Complexity

Linear in the size of the container.

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=cpp/container/unordered\\_multimap/%7Eunordered\\_multimap&oldid=50739](http://en.cppreference.com/mwiki/index.php?title=cpp/container/unordered_multimap/%7Eunordered_multimap&oldid=50739)"

## std::unordered\_multimap::operator=

<code>unordered_multimap&amp; operator=( const unordered_multimap&amp; other );</code>	(1)	(since C++11)
<code>unordered_multimap&amp; operator=( unordered_multimap&amp;&amp; other );</code>	(2)	(since C++11)
<code>unordered_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<Hash>::value  
&& std::is_nothrow_move_assignable<Pred>::value)`

### Example

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

Run this code

```

#include <unordered_map>
#include <iostream>

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

int main()
{
    std::unordered_multimap<int, int> nums1 {{3, 1}, {4, 1}, {5, 9},
                                             {6, 1}, {7, 1}, {8, 9}};
    std::unordered_multimap<int, int> nums2;
    std::unordered_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 `unordered_multimap`  
 (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/operator%3D&oldid=41544"



## std::unordered\_multimap::get\_allocator

---

```
allocator_type get_allocator() const;    (since C++11)
```

---

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/unordered\\_multimap/get\\_allocator&oldid=50725](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/get_allocator&oldid=50725)"

## std::unordered\_multimap::begin, std::unordered\_multimap::cbegin

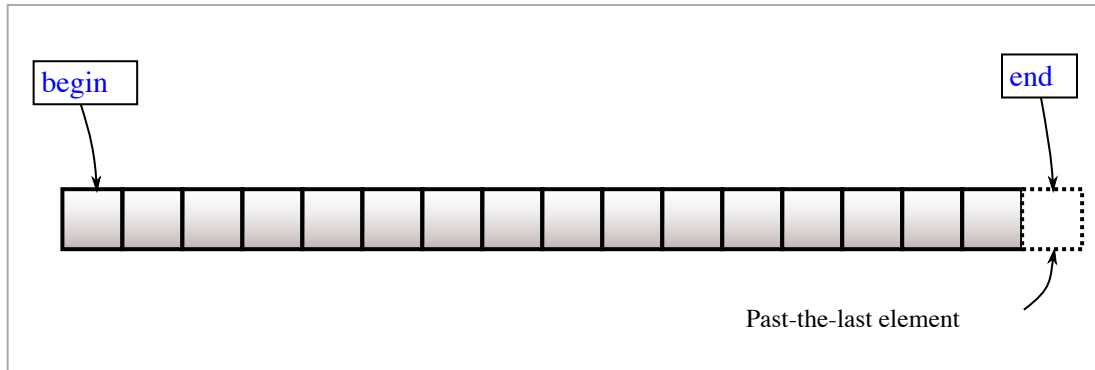
---

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

noexcept specification: [noexcept](#)

### 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=c++/container/unordered\_multimap/begin&oldid=50710"

## std::unordered\_multimap::end, std::unordered\_multimap::cend

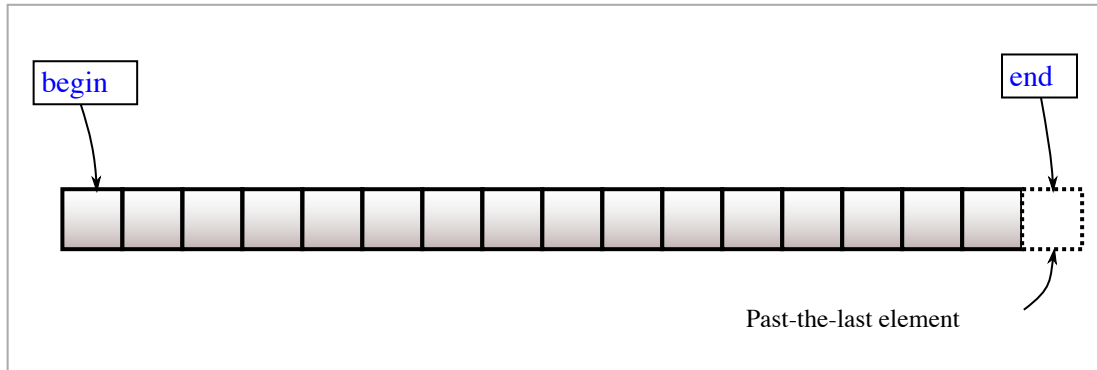
---

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

noexcept specification: [noexcept](#)

### 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=c++/container/unordered\_multimap/end&oldid=50720"

## std::unordered\_multimap::erase

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

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 iterators and references are not invalidated.

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.

The order of the elements that are not erased is preserved (this makes it possible to erase individual elements while iterating through the container) (since C++14)

### 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 unordered\_multimap:

- 1) Average case: constant, worst case: `c.size()`
- 2) Average case: `std::distance(first, last)`, worst case: `c.size()`
- 3) Average case: `c.count(key)`, worst case: `c.size()`

### Example

Run this code

```
#include <unordered_map>
#include <iostream>
int main()
{
    std::unordered_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/unordered\_multimap/erase&oldid=50723"

## std::unordered\_multimap::size

---

```
size_type size() const;    (since C++11)
```

---

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

noexcept specification: [noexcept](#)

### Complexity

Constant.

### Example

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

Run this code

```
#include <unordered_map>
#include <iostream>

int main()
{
    std::unordered_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/unordered\_multimap/size&oldid=50735"

## std::unordered\_multimap::max\_size

---

```
size_type max_size() const;    (since C++11)
```

---

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

noexcept specification: [`noexcept`](#)

### 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 <unordered_multimap>

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

Possible output:

```
Maximum size of a 'unordered_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/unordered\_multimap/max\_size&oldid=50731"

## std::unordered\_multimap::clear

---

```
void clear();
```

(since C++11)

---

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

noexcept specification: noexcept

### Complexity

Linear in the size of the container.

### See also

---

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

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/clear&oldid=50715"



## std::unordered\_multimap::insert

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

Inserts element(s) into the container.

- 1-2) inserts value. The overload (2) is equivalent to `emplace(std::forward<P>(value))` and only participates in overload resolution if `std::is_constructible<value_type, P&&>::value == true`.
- 3-4) inserts value, using hint as a non-binding suggestion to where the search should start. 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.

If rehashing occurs due to the insertion, all iterators are invalidated. Otherwise iterators are not affected. References are not invalidated. Rehashing occurs only if the new number of elements is equal to or greater than `max_load_factor()*bucket_count()`.

### Parameters

- hint** - iterator, used as a suggestion as to where to insert the content
- value** - element value to insert
- first, last** - range of elements to insert
- ilist** - initializer list to insert the values from

### Type requirements

- InputIt must meet the requirements of InputIterator.

### Return value

- 1-4) 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

- 1-4) Average case:  $O(1)$ , worst case  $O(\text{size}())$
- 5-6) Average case:  $O(N)$ , where  $N$  is the number of elements to insert. Worst case:  $O(N * \text{size}() + N)$

See also

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

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/insert&oldid=74443"

## std::unordered\_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)...`.

If rehashing occurs due to the insertion, all iterators are invalidated. Otherwise iterators are not affected. References are not invalidated. Rehashing occurs only if the new number of elements is equal to or greater than `max_load_factor()*bucket_count()`.

### 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

Amortized constant on average, worst case linear in the size of the container.

### Example

Run this code

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

#include <unordered_map>
int main()
{
    std::unordered_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';
    }
}
```

Possible output:

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

## See also

<b>emplace_hint</b>	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/unordered\\_multimap/emplace&oldid=50717](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/emplace&oldid=50717)"

## std::unordered\_multimap::emplace\_hint

---

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

---

Inserts a new element to the container, using `hint` as a suggestion where the element should go. 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)...`.

If rehashing occurs due to the insertion, all iterators are invalidated. Otherwise iterators are not affected. References are not invalidated. Rehashing occurs only if the new number of elements is equal to or greater than `max_load_factor()*bucket_count()`.

### Parameters

**hint** - iterator, used as a suggestion as to where to insert the new element  
**args** - arguments to forward to the constructor of the element

### Return value

Returns an iterator to the newly inserted element.

### Complexity

Amortized constant on average, worst case linear in the size of the container.

### See also

---

<b>emplace</b>	constructs element in-place <span style="color: green;">(public member function)</span>
<b>insert</b>	inserts elements <span style="color: green;">(public member function)</span>

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/emplace\_hint&oldid=65117"

---

## std::unordered\_multimap::swap

---

```
void swap( unordered_multimap& other );    (since C++11)
```

---

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 `Hash` and `KeyEqual` objects must be `Swappable`, and they are exchanged using unqualified calls 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 `Hash` or `KeyEqual` objects. (until C++17)

noexcept specification:

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value
&& noexcept(std::swap(std::declval<Hash&>(), std::declval<Hash&>()))    (since C++17)
&& noexcept(std::swap(std::declval<KeyEqual&>(), std::declval<KeyEqual&>()))
```

### Complexity

Constant.

### See also

---

**std::swap**(std::unordered\_multimap) (C++11) specializes the `std::swap` algorithm (function template)

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/swap&oldid=50736"

## std::unordered\_multimap::count

---

```
size_type count( const Key& key ) const;    (1) (since C++11)
```

---

Returns the number of elements with key `key`.

### Parameters

**key** - key value of the elements to count

### Return value

Number of elements with key `key`.

### Complexity

linear in the number of elements with key `key` on average, worst case linear in the size of the container.

### 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/unordered\\_multimap/count&oldid=65129](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/count&oldid=65129)"

## std::unordered\_multimap::find

---

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

---

(1)

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

---

(2)

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

### Parameters

**key** - key value of the element to search for

### 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

Constant on average, worst case linear in the size of the container.

### Example

Run this code

```
#include <iostream>
#include <unordered_map>

int main()
{
    std::unordered_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/unordered\_multimap/find&oldid=50724"



## std::unordered\_multimap::equal\_range

---

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

(since C++11)

---

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

(since C++11)

---

Returns a range containing all elements with key `key` in the container. The range is defined by two iterators, the first pointing to the first element of the wanted range and the second pointing past the last element of the range.

### Parameters

**key** - key value to compare the elements to

### Return value

`std::pair` containing a pair of iterators defining the wanted range. If there are no such elements, past-the-end (see `end()`) iterators are returned as both elements of the pair.

### Complexity

Average case constant, worst case linear in the size of the container.

### Example

Run this code

```
#include <iostream>
#include <unordered_map>

int main()
{
    std::unordered_multimap<int,char> map = {{1,'a'}, {1,'b'}, {1,'d'}, {2,'b'}};
    auto range = map.equal_range(1);
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->first << ' ' << it->second << '\n';
    }
}
```

Output:

```
1 a
1 b
1 d
```

### See also

**find** finds element with specific key  
(public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/equal\_range&oldid=65096"

## std::unordered\_multimap::begin(int), std::unordered\_multimap::cbegin(int)

---

local_iterator begin( size_type n );	(since C++11)
const_local_iterator begin( size_type n ) const;	(since C++11)
const_local_iterator cbegin( size_type n ) const;	(since C++11)

---

Returns an iterator to the first element of the bucket with index `pos`.

### Parameters

**n** - the index of the bucket to access

### Return value

Iterator to the first element.

### Complexity

Constant.

### See also

---

<b>end</b> (int)	returns an iterator to the end of the specified bucket
<b>end</b> (int)	(public member function)

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\\_multimap/begin2&oldid=50711](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/begin2&oldid=50711)"

## std::unordered\_multimap::end(int), std::unordered\_multimap::cend(int)

---

local_iterator end( size_type n );	(since C++11)
const_local_iterator end( size_type n ) const;	(since C++11)
const_local_iterator cend( size_type n ) const;	(since C++11)

---

Returns an iterator to the element following the last element of the bucket with index `n`. . This element acts as a placeholder, attempting to access it results in undefined behavior.

### Parameters

**n** - the index of the bucket to access

### Return value

iterator to the element following the last element

### Complexity

Constant

### See also

---

**begin**(int) returns an iterator to the beginning of the specified bucket  
**cbegin**(int) (public member function)

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/end2&oldid=50721"

---

## std::unordered\_multimap::bucket\_count

---

```
size_type bucket_count ( ) const; (since C++11)
```

---

Returns the number of buckets in the container.

### Parameters

(none)

### Return value

The number of buckets in the container.

### Complexity

Constant.

### See also

<b>bucket_size</b>	returns the number of elements in specific bucket <span style="color: green;">(public member function)</span>
<b>max_bucket_count</b>	returns the maximum number of buckets <span style="color: green;">(public member function)</span>

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\\_multimap/bucket\\_count&oldid=50713](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/bucket_count&oldid=50713)"

## std::unordered\_multimap::max\_bucket\_count

---

```
size_type max_bucket_count() const;    (since C++11)
```

---

Returns the maximum number of buckets the container is able to hold due to system or library implementation limitations.

### Parameters

(none)

### Return value

Maximum number of buckets.

### Complexity

Constant.

### See also

---

<b>bucket_count</b>	returns the number of buckets (public member function)
---------------------	---

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/max\_bucket\_count&oldid=50729"

## std::unordered\_multimap::bucket\_size

---

```
size_type bucket_size( size_type n ) const; (since C++11)
```

---

Returns the number of elements in the bucket with index *n*.

### Parameters

**n** - the index of the bucket to examine

### Return value

The number of elements in the bucket *n*.

### Complexity

Constant.

### See also

---

<b>bucket_count</b>	returns the number of buckets <span style="color: green;">(public member function)</span>
---------------------	--

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/bucket\_size&oldid=50714"

## std::unordered\_multimap::bucket

---

```
size_type bucket( const Key& key ) const;    (since C++11)
```

---

Returns the index of the bucket for key `key`. Elements (if any) with keys equivalent to `key` are always found in this bucket. The returned value is valid only for instances of the container for which `bucket_count()` returns the same value.

The behavior is undefined if `bucket_count()` is zero.

### Parameters

**key** - the value of the key to examine

### Return value

Bucket index for the key `key`.

### Complexity

Constant.

### See also

---

**bucket\_size** returns the number of elements in specific bucket  
(public member function)

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/bucket&oldid=50712"

## std::unordered\_multimap::load\_factor

---

```
float load_factor() const;    (since C++11)
```

---

Returns the average number of elements per bucket.

### Parameters

(none)

### Return value

Average number of elements per bucket.

### Complexity

Constant.

### See also

---

<b>max_load_factor</b>	manages maximum average number of elements per bucket (public member function)
------------------------	---

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\\_multimap/load\\_factor&oldid=50728](http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered_multimap/load_factor&oldid=50728)"



## std::unordered\_multimap::max\_load\_factor

---

<code>float max_load_factor() const;</code>	(1)	(since C++11)
<code>void max_load_factor( float ml );</code>	(2)	(since C++11)

---

Manages the maximum load factor (number of elements per bucket). The container automatically increases the number of buckets if the load factor exceeds this threshold.

- 1) Returns current maximum load factor.
- 2) Sets the maximum load factor to `ml`.

### Parameters

**ml** - new maximum load factor setting

### Return value

- 1) current maximum load factor.
- 2) none.

### Complexity

Constant

### See also

---

<b>load_factor</b>	returns average number of elements per bucket (public member function)
--------------------	---

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/max\_load\_factor&oldid=50730"

## std::unordered\_multimap::rehash

---

```
void rehash( size_type count );    (since C++11)
```

---

Sets the number of buckets to `count` and rehashes the container, i.e. puts the elements into appropriate buckets considering that total number of buckets has changed. If the new number of buckets makes load factor more than maximum load factor (`count < size() / max_load_factor()`), then the new number of buckets is at least `size() / max_load_factor()`.

### Parameters

**count** - new number of buckets

### Return value

(none)

### Complexity

Average case linear in the size of the container, worst case quadratic.

### Notes

`rehash(0)` may be used to force an unconditional rehash, such as after suspension of automatic rehashing by temporarily increasing `max_load_factor()`.

### See also

---

<b>reserve</b>	reserves space for at least the specified number of elements. This regenerates the hash table. (public member function)
----------------	---

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/rehash&oldid=50733"

## std::unordered\_multimap::reserve

---

```
void reserve( size_type count );    (since C++11)
```

---

Sets the number of buckets to the number needed to accomodate at least `count` elements without exceeding maximum load factor and rehashes the container, i.e. puts the elements into appropriate buckets considering that total number of buckets has changed. Effectively calls

```
rehash(std::ceil(count / max_load_factor())).
```

### Parameters

**count** - new capacity of the container

### Return value

(none)

### Complexity

Average case linear in the size of the container, worst case quadratic.

### See also

---

<b>rehash</b>	reserves at least the specified number of buckets. This regenerates the hash table. (public member function)
---------------	--

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/reserve&oldid=50734"

## std::unordered\_multimap::hash\_function

---

```
hasher hash_function( ) const; (since C++11)
```

---

Returns the function that hashes the keys.

### Parameters

(none)

### Return value

The hash function.

### Complexity

Constant.

### See also

---

**key\_eq** returns the function used to compare keys for equality  
(public member function)

---

Retrieved from "[http://en.cppreference.com/mwiki/index.php?title=cpp/container/unordered\\_multimap/hash\\_function&oldid=50726](http://en.cppreference.com/mwiki/index.php?title=cpp/container/unordered_multimap/hash_function&oldid=50726)"

## std::unordered\_multimap::key\_eq

---

```
key_equal key_eq() const;    (since C++11)
```

---

Returns the function that compares keys for equality.

### Parameters

(none)

### Return value

The key comparison function.

### Complexity

Constant.

### See also

---

<b>hash_function</b>	returns function used to hash the keys (public member function)
----------------------	--

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/key\_eq&oldid=50727"

## std::unordered\_multimap::empty

---

```
bool empty() const;    (since C++11)
```

---

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

noexcept specification: `noexcept`

### Complexity

Constant.

### Example

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

Run this code

```
#include <unordered_map>
#include <iostream>
#include <utility>

int main()
{
    std::unordered_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/unordered\_multimap/empty&oldid=50719"

# operator==,!=(std::unordered\_multimap)

---

```
template< class Key, class T, class Hash, class KeyEqual, class Allocator >
bool operator==( const unordered_multimap<Key,T,Hash,KeyEqual,Allocator>& lhs,           (1)
                 const unordered_multimap<Key,T,Hash,KeyEqual,Allocator>& rhs );

template< class Key, class T, class Hash, class KeyEqual, class Allocator >
bool operator!=( const unordered_multimap<Key,T,Hash,KeyEqual,Allocator>& lhs,           (2)
                 const unordered_multimap<Key,T,Hash,KeyEqual,Allocator>& rhs );
```

---

Compares the contents of two unordered containers.

The contents of two unordered containers `lhs` and `rhs` are equal if the following conditions hold:

- `lhs.size() == rhs.size()`
- each group of equivalent keys `[lhs_eq1, lhs_eq2)` obtained from `lhs.equal_range(lhs_eq1)` has a corresponding group of equivalent keys in the other container `[rhs_eq1, rhs_eq2)` obtained from `rhs.equal_range(rhs_eq1)`, that has the following properties:
  - `std::distance(lhs_eq1, lhs_eq2) == std::distance(rhs_eq1, rhs_eq2)`.
  - `std::is_permutation(lhs_eq1, lhs_eq2, rhs_eq1) == true`.

The behavior is undefined if `Key` or `T` are not `EqualityComparable`.

The behavior is also undefined if `Hash` and `KeyEqual` do not have the same behavior on `lhs` and `rhs` or if the equality comparison operator for `Key` is not a refinement of the partition into equivalent-key groups introduced by `KeyEqual` (that is, if two keys that compare equal fall into different partitions)

## Parameters

**lhs, rhs** - unordered containers to compare

## 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

## Complexity

$\Sigma S_i^2$  comparisons of the keys in the average case, where  $S$  is the size of the  $i$ th equivalent key group.  $N^2$  comparisons of the keys in the worst case, where  $N$  is the size of the container.

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/operator\_cmp&oldid=50732"

## std::swap(std::unordered\_multimap)

---

```
template< class Key, class T, class Hash, class KeyEqual, class Alloc >      (since  
void swap( unordered_multimap<Key,T,Hash,KeyEqual,Alloc>& lhs,              C++11)  
           unordered_multimap<Key,T,Hash,KeyEqual,Alloc>& rhs );
```

---

Specializes the `std::swap` algorithm for `std::unordered_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:

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

(since C++17)

### See also

---

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

---

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=c++/container/unordered\_multimap/swap2&oldid=50737"