std::unordered_map

Defined in header <unordered_map>

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

Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements 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.

`std::unordered_map` meets the requirements of `Container`, `AllocatorAwareContainer`, `UnorderedAssociativeContainer`.

Iterator invalidation

Operations	Invalidated
All read only operations, `swap`, `std::swap`	Never
`clear`, `rehash`, `reserve`, `operator=`	Always
`insert`, `emplace`, `emplace_hint`, `operator[]`	Only if causes rehash
`erase`	Only to the element erased

Notes

- The swap functions do not invalidate any of the iterators inside the container, but they do invalidate the iterator marking the end of the swap region.
- References and pointers to either key or data stored in the container are only invalidated by erasing that element, even when the corresponding iterator is invalidated.

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_map` (public member function)
(destructor)	destructs the `unordered_map` (public member function)
`operator=`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Iterators

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

Capacity

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

Modifiers

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

Lookup

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`count`	returns the number of elements matching specific key (public member function)
`find`	finds element with specific key (public member function)
`equal_range`	returns range of elements matching a specific key (public member function)

Bucket interface

`begin``(int)` **`cbegin`**`(int)`	returns an iterator to the beginning of the specified bucket (public member function)
`end``(int)` **`cend`**`(int)`	returns an iterator to the end of the specified bucket (public member function)
`bucket_count`	returns the number of buckets (public member function)
`max_bucket_count`	returns the maximum number of buckets (public member function)

bucket_size	returns the number of elements in specific bucket (public member function)
bucket	returns the bucket for specific key (public member function)

Hash policy

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

Observers

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

Non-member functions

operator== **operator!=**	compares the values in the unordered_map (function template)
std::swap(std::unordered_map) (C++11)	specializes the `std::swap` algorithm (function template)

std::unordered_map::unordered_map

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

Run this code

```
#include <unordered_map>
#include <vector>
#include <bitset>
#include <string>
#include <utility>

struct Key {
    std::string first;
    std::string second;
};

struct KeyHash {
 std::size_t operator()(const Key& k) const
 {
     return std::hash<std::string>()(k.first) ^
            (std::hash<std::string>()(k.second) << 1);
 }
};

struct KeyEqual {
 bool operator()(const Key& lhs, const Key& rhs) const
 {
    return lhs.first == rhs.first && lhs.second == rhs.second;
 }
```

```
};

int main()
{
    // default constructor: empty map
    std::unordered_map<std::string, std::string> m1;

    // list constructor
    std::unordered_map<int, std::string> m2 =
    {
        {1, "foo"},
        {3, "bar"},
        {2, "baz"},
    };

    // copy constructor
    std::unordered_map<int, std::string> m3 = m2;

    // move constructor
    std::unordered_map<int, std::string> m4 = std::move(m2);

    // range constructor
    std::vector<std::pair<std::bitset<8>, int>> v = { {0x12, 1}, {0x01,-1} };
    std::unordered_map<std::bitset<8>, double> m5(v.begin(), v.end());

    // constructor for a custom type
    std::unordered_map<Key, std::string, KeyHash, KeyEqual> m6 = {
            { {"John", "Doe"}, "example"},
            { {"Mary", "Sue"}, "another"}
    };
}
```

See also

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

std::unordered_map::~unordered_map

```
~unordered_map();    (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.

std::unordered_map::operator=

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

Replaces the contents of the container.

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

Parameters

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

Return value

`*this`

Complexity

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

Exceptions

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_map` to another:

Run this code

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

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

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

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

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

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

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

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

Output:

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

See also

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

std::unordered_map::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

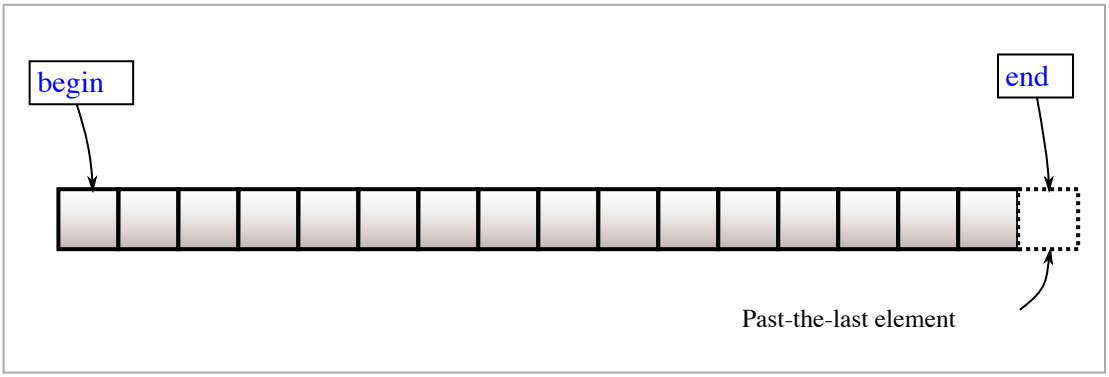
Constant.

std::unordered_map::begin, std::unordered_map::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

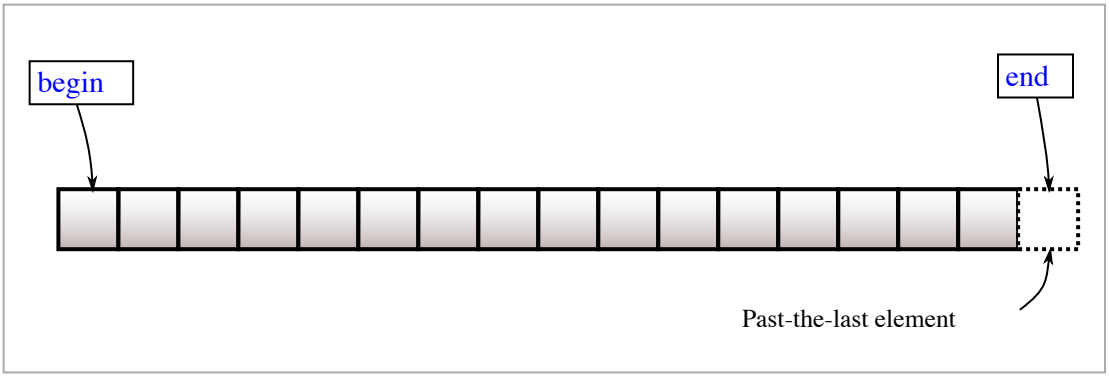
end **cend**	returns an iterator to the end (public member function)

std::unordered_map::end, std::unordered_map::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

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

std::unordered_map::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 the element (if one exists) 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 dereferencable) 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_map`:

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_map<int, std::string> c = {{1, "one"}, {2, "two"}, {3, "three"},
                                              {4, "four"}, {5, "five"}, {6, "six"}};
    // erase all odd numbers from c
    for(auto it = c.begin(); it != c.end(); )
        if(it->first % 2 == 1)
```

```
            it = c.erase(it);
        else
            ++it;
    for(auto& p : c)
        std::cout << p.second << ' ';
}
```

Output:

```
two four six
```

See also

clear	clears the contents (public member function)

std::unordered_map::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_map`:

Run this code

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

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

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

Output:

```
nums contains 4 elements.
```

See also

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

std::unordered_map::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_map>

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

Possible output:

```
Maximum size of a 'unordered_map' is 1844674407370955161 5
```

See also

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

std::unordered_map::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

erase	erases elements (public member function)

std::unordered_map::insert

`std::pair<iterator,bool> insert( const value_type& value );`	(1)	(since C++11)
`template< class P >` `std::pair<iterator,bool> insert( P&& value );`	(2)	(since C++11)
`std::pair<iterator,bool> insert( value_type&& value );`	(2)	(since C++17)
`iterator insert( 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, if the container doesn't already contain an element with an equivalent key.

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

3-4) Inserts `value`, 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

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

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

5-6) (none)

Exceptions

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

This section is incomplete
Reason: cases 5-6

Complexity

1-4) Average case: `O(1)`, worst case `O(size())`

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

Example

Run this code

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

int main ()
{
    std::unordered_map<int, std::string> dict = {{1, "one"}, {2, "two"}};
    dict.insert({3, "three"});
    dict.insert(std::make_pair(4, "four"));
    dict.insert({{4, "another four"}, {5, "five"}});

    bool ok = dict.insert({1, "another one"}).second;
    std::cout << "inserting 1 -> \"another one\" "
              << (ok ? "succeeded" : "failed") << '\n';

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

Possible output:

```
inserting 1 -> "another one" failed
contents:
 5 => five
 1 => one
 2 => two
 3 => three
 4 => four
```

See also

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

std::unordered_map::insert_or_assign

`template <class M>` `pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);`	(1)	(since C++17)
`template <class M>` `pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);`	(2)	(since C++17)
`template <class M>` `iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);`	(3)	(since C++17)
`template <class M>` `iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);`	(4)	(since C++17)

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

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

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

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

Return value

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

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

Complexity

1,2) Same as for emplace

3,4) Same as for emplace_hint

Notes

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

Example

This section is incomplete
Reason: no example

See also

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

std::unordered_map::emplace

```
template< class... Args >
std::pair<iterator,bool> emplace( Args&&... args );
```
(since C++11)

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

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

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

Exceptions

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

Complexity

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_map<std::string, std::string> m;

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

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

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

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

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

```
}
```

Possible output:

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

See also

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

std::unordered_map::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.

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

Complexity

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

See also

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

std::unordered_map::try_emplace

`template <class... Args>` `pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);`	(1)	(since C++17)
`template <class... Args>` `pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);`	(2)	(since C++17)
`template <class... Args>` `iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);`	(3)	(since C++17)
`template <class... Args>` `iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);`	(4)	(since C++17)

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

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

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

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

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

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

Return value

1,2) Same as for emplace

3,4) Same as for emplace_hint

Complexity

1,2) Same as for emplace

3,4) Same as for emplace_hint

Notes

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

Example

This section is incomplete
Reason: no example

See also

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

std::unordered_map::swap

```
void swap( unordered_map& 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, they are not swapped (and if `get_allocator() != other.get_allocator()`, the behavior is undefined). (since C++11)

Parameters

other - container to exchange the contents with

Return value

(none)

Exceptions

Any exception thrown by the swap `Hash` or `KeyEqual` objects. (until C++17)

`noexcept` specification: (since C++17)

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

Complexity

Constant.

See also

std::swap(std::unordered_map) (C++11)	specializes the `std::swap` algorithm (function template)

std::unordered_map::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

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

See also

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

std::unordered_map::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_map<int,char> example = {{1,'a'},{2,'b'}};

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

Output:

```
Found 2 b
```

See also

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

std::unordered_map::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_map<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
```

See also

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

std::unordered_map::begin(int), std::unordered_map::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

end`(int)`	returns an iterator to the end of the specified bucket
cend`(int)`	(public member function)

std::unordered_map::end(int), std::unordered_map::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)

std::unordered_map::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

bucket_size	returns the number of elements in specific bucket (public member function)
max_bucket_count	returns the maximum number of buckets (public member function)

std::unordered_map::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

bucket_count	returns the number of buckets (public member function)

std::unordered_map::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

bucket_count	returns the number of buckets (public member function)

std::unordered_map::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)

std::unordered_map::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

`max_load_factor`	manages maximum average number of elements per bucket (public member function)

std::unordered_map::max_load_factor

`float max_load_factor() const;`	(1)	(since C++11)
`void max_load_factor( float ml );`	(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

`load_factor`	returns average number of elements per bucket (public member function)

std::unordered_map::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

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

std::unordered_map::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

rehash	reserves at least the specified number of buckets. This regenerates the hash table. (public member function)

std::unordered_map::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)

std::unordered_map::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

`hash_function`	returns function used to hash the keys (public member function)

std::unordered_map::operator[]

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

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

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

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

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

If an insertion occurs and results in a rehashing of the container, 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

`key` - the key of the element to find

Return value

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

Exceptions

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

Complexity

Average case: constant, worst case: linear in size.

Example

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

Run this code

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

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

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

    letter_counts['b'] = 42;  // update an existing value

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

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

Output:

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

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

Run this code

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

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

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

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

Output:

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

See also

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

std::unordered_map::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_map<int,int>` contains any elements:

Run this code

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

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

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

Output:

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

See also

size	returns the number of elements (public member function)

std::unordered_map::at

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

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

Parameters

`key` - the key of the element to find

Return value

Reference to the mapped value of the requested element

Exceptions

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

Complexity

Average case: constant, worst case: linear in size.

See also

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

operator==,!=(std::unordered_map)

```
template< class Key, class T, class Hash, class KeyEqual, class Allocator >
bool operator==( const unordered_map<Key,T,Hash,KeyEqual,Allocator>& lhs,      (1)
                 const unordered_map<Key,T,Hash,KeyEqual,Allocator>& rhs );
template< class Key, class T, class Hash, class KeyEqual, class Allocator >
bool operator!=( const unordered_map<Key,T,Hash,KeyEqual,Allocator>& lhs,      (2)
                 const unordered_map<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

N comparisons of the keys in the average case, N^2 in the worst case, where *N* is the size of the container.

std::swap(std::unordered_map)

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

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

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Constant.

Exceptions

(since C++17)

`noexcept` specification:

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

See also

`swap`	swaps the contents (public member function)