

This is the documentation for an old version of Boost. Click here to view this page for the latest version.

e world."

andrescu

([http://en.wikipedia.org/wiki/Andrei\\_Alexandrescu](http://en.wikipedia.org/wiki/Andrei_Alexandrescu)), C++ Coding Standards

([https://books.google.com/books/about/C%2B%2B\\_Coding\\_Standards.html?id=mmjVIC6WolgC](https://books.google.com/books/about/C%2B%2B_Coding_Standards.html?id=mmjVIC6WolgC))

# Boost.Unordered

---

## Table of Contents

Introduction

Basics of Hash Tables

    Controlling the Number of Buckets

Equality Predicates and Hash Functions

    Custom Types

Regular Containers

    Iterator Invalidation

    Comparison with Associative Containers

Concurrent Containers

    Visitation-based API

    Whole-Table Visitation

    Bulk visitation

    Blocking Operations

    Interoperability with non-concurrent containers

Standard Compliance

    Closed-addressing Containers

        Deduction Guides

        Piecewise Pair Emplacement

        Swap

    Open-addressing Containers

    Concurrent Containers

Data Structures

    Closed-addressing Containers

    Open-addressing Containers

    Concurrent Containers

Benchmarks

    boost::unordered\_[multi]set

        GCC 12 + libstdc++-v3, x64

        Clang 15 + libc++, x64

        Visual Studio 2022 + Dinkumware, x64

    boost::unordered\_(flat|node)\_map

        GCC 12, x64

        Clang 15, x64

        Visual Studio 2022, x64

        Clang 12, ARM64

GCC 12, x86  
Clang 15, x86  
Visual Studio 2022, x86  
boost::concurrent\_flat\_map

GCC 12, x64  
Clang 15, x64  
Visual Studio 2022, x64  
Clang 12, ARM64  
GCC 12, x86  
Clang 15, x86  
Visual Studio 2022, x86

## Implementation Rationale

Closed-addressing Containers

    Data Structure  
    Number of Buckets

Open-addressing Containers

    Hash Function  
    Platform Interoperability

Concurrent Containers

    Hash Function and Platform Interoperability

## Reference

Class Template unordered\_map

    Synopsis  
    Description  
    Configuration macros

    Typedefs

    Constructors

    Destructor

    Assignment

    Iterators

    Size and Capacity

    Modifiers

    Observers

    Lookup

    Bucket Interface

    Hash Policy

    Deduction Guides

    Equality Comparisons

    Swap

    erase\_if

    Serialization

Class Template unordered\_multimap

    Synopsis  
    Description  
    Configuration macros

Typedefs  
Constructors  
Destructor  
Assignment  
Iterators  
Size and Capacity

Modifiers  
Observers  
Lookup  
Bucket Interface  
Hash Policy  
Deduction Guides  
Equality Comparisons  
Swap  
erase\_if  
Serialization

#### Class Template unordered\_set

Synopsis  
Description  
Configuration macros  
Typedefs  
Constructors  
Destructor  
Assignment  
Iterators  
Size and Capacity  
Modifiers  
Observers  
Lookup  
Bucket Interface  
Hash Policy  
Deduction Guides  
Equality Comparisons  
Swap  
erase\_if  
Serialization

#### Class Template unordered\_multiset

Synopsis  
Description  
Configuration macros  
Typedefs  
Constructors  
Destructor  
Assignment  
Iterators

Size and Capacity

Modifiers

Observers

Lookup

Bucket Interface

Hash Policy

Deduction Guides

Equality Comparisons

Swap

erase\_if

Serialization

Hash traits

Synopsis

hash\_is\_avalanching

Class Template unordered\_flat\_map

Synopsis

Description

Typedefs

Constructors

Destructor

Assignment

Iterators

Size and Capacity

Modifiers

Observers

Lookup

Bucket Interface

Hash Policy

Deduction Guides

Equality Comparisons

Swap

erase\_if

Serialization

Class Template unordered\_flat\_set

Synopsis

Description

Typedefs

Constructors

Destructor

Assignment

Iterators

Size and Capacity

Modifiers

Observers

Lookup

Bucket Interface  
Hash Policy  
Deduction Guides  
Equality Comparisons  
Swap  
erase\_if  
Serialization

Class Template unordered\_node\_map

Synopsis  
Description  
Typedefs  
Constructors  
Destructor  
Assignment  
Iterators  
Size and Capacity  
Modifiers  
Observers  
Lookup  
Bucket Interface  
Hash Policy  
Deduction Guides  
Equality Comparisons  
Swap  
erase\_if  
Serialization

Class Template unordered\_node\_set

Synopsis  
Description  
Typedefs  
Constructors  
Destructor  
Assignment  
Iterators  
Size and Capacity  
Modifiers  
Observers  
Lookup  
Bucket Interface  
Hash Policy  
Deduction Guides  
Equality Comparisons  
Swap  
erase\_if  
Serialization

## Class Template concurrent\_flat\_map

Synopsis

Description

Concurrency Requirements and Guarantees

Configuration Macros

Constants

Constructors

Destructor

Assignment

Visitation

Size and Capacity

Modifiers

Observers

Map Operations

Bucket Interface

Hash Policy

Deduction Guides

Equality Comparisons

Swap

erase\_if

Serialization

## Class Template concurrent\_flat\_set

Synopsis

Description

Concurrency Requirements and Guarantees

Configuration Macros

Constants

Constructors

Destructor

Assignment

Visitation

Size and Capacity

Modifiers

Observers

Set Operations

Bucket Interface

Hash Policy

Deduction Guides

Equality Comparisons

Swap

erase\_if

Serialization

## Change Log

Release 1.84.0 - Major update

Release 1.83.0 - Major update

Release 1.82.0 - Major update

Release 1.81.0 - Major update

Release 1.80.0 - Major update

Release 1.79.0

Release 1.67.0

Release 1.66.0

Release 1.65.0

Release 1.64.0

Release 1.63.0

Release 1.62.0

Release 1.58.0

Release 1.57.0

Release 1.56.0

Release 1.55.0

Release 1.54.0

Release 1.53.0

Release 1.52.0

Release 1.51.0

Release 1.50.0

Release 1.49.0

Release 1.48.0 - Major update

Release 1.45.0

Release 1.43.0

Release 1.42.0

Release 1.41.0 - Major update

Release 1.40.0

Release 1.39.0

Release 1.38.0

Release 1.37.0

Release 1.36.0

Boost 1.35.0 Add-on - 31st March 2008

Review Version

Bibliography

Copyright and License

---

## Introduction

Hash tables ([https://en.wikipedia.org/wiki/Hash\\_table](https://en.wikipedia.org/wiki/Hash_table)) are extremely popular computer data structures and can be found under one form or another in virtually any programming language. Whereas other associative structures such as rb-trees (used in C++ by `std::set` and `std::map`) have logarithmic-time complexity for insertion and lookup, hash tables, if configured properly, perform these operations in constant time on average, and are generally much faster.

C++ introduced *unordered associative containers* `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` and `std::unordered_multimap` in C++11, but research on hash tables hasn't stopped since: advances in CPU architectures such as more powerful caches, SIMD

([https://en.wikipedia.org/wiki/Single\\_instruction,\\_multiple\\_data](https://en.wikipedia.org/wiki/Single_instruction,_multiple_data)) operations and increasingly available [multicore processors](#) ([https://en.wikipedia.org/wiki/Multi-core\\_processor](https://en.wikipedia.org/wiki/Multi-core_processor)) open up possibilities for improved hash-based data structures and new use cases that are simply beyond reach of unordered associative containers as specified in 2011. Boost.Unordered offers a catalog of hash containers with different standards compliance levels, performances and intended usage scenarios:

*Table 1. Boost.Unordered containers*

	<b>Node-based</b>	<b>Flat</b>
<b>Closed addressing</b>	<code>boost::unordered_set</code> <code>boost::unordered_map</code> <code>boost::unordered_multiset</code> <code>boost::unordered_multimap</code>	
<b>Open addressing</b>	<code>boost::unordered_node_set</code> <code>boost::unordered_node_map</code>	<code>boost::unordered_flat_set</code> <code>boost::unordered_flat_map</code>
<b>Concurrent</b>		<code>boost::concurrent_flat_set</code> <code>boost::concurrent_flat_map</code>

- **Closed-addressing containers** are fully compliant with the C++ specification for unordered associative containers and feature one of the fastest implementations in the market within the technical constraints imposed by the required standard interface.
- **Open-addressing containers** rely on much faster data structures and algorithms (more than 2 times faster in typical scenarios) while slightly diverging from the standard interface to accommodate the implementation. There are two variants: **flat** (the fastest) and **node-based**, which provide pointer stability under rehashing at the expense of being slower.
- Finally, **concurrent containers** are designed and implemented to be used in high-performance multithreaded scenarios. Their interface is radically different from that of regular C++ containers.

All sets and maps in Boost.Unordered are instantiated similarly as `std::unordered_set` and `std::unordered_map`, respectively:

```

namespace boost {
    template <
        class Key,
        class Hash = boost::hash<Key>,
        class Pred = std::equal_to<Key>,
        class Alloc = std::allocator<Key> >
    class unordered_set;
    // same for unordered_multiset, unordered_flat_set, unordered_node_set
    // and concurrent_flat_set

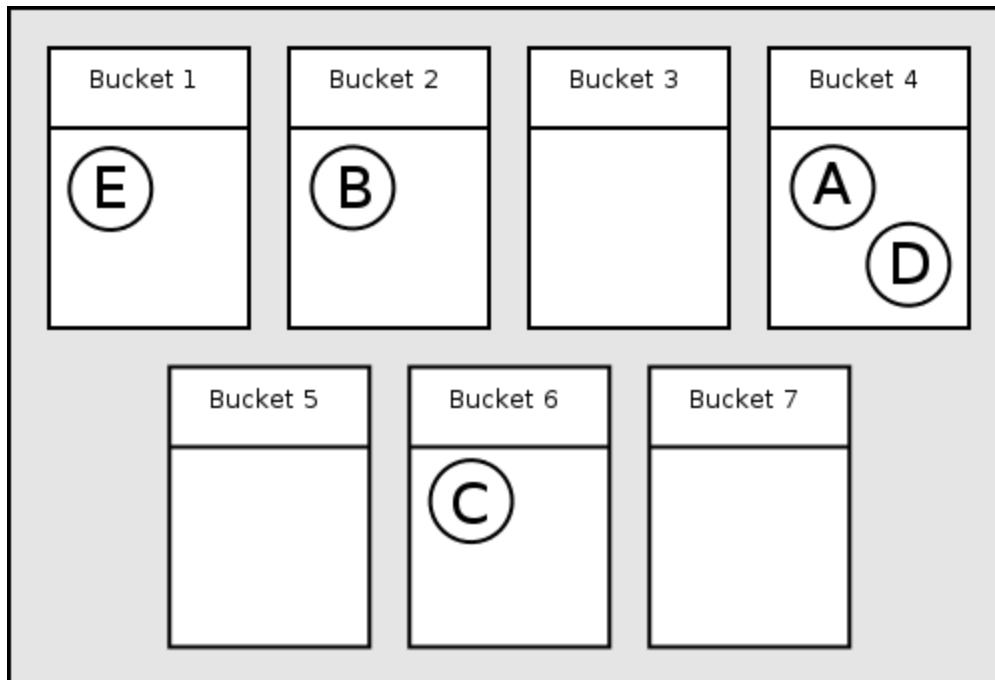
    template <
        class Key, class Mapped,
        class Hash = boost::hash<Key>,
        class Pred = std::equal_to<Key>,
        class Alloc = std::allocator<std::pair<Key const, Mapped> > >
    class unordered_map;
    // same for unordered_multimap, unordered_flat_map, unordered_node_map
    // and concurrent_flat_map
}

```

Storing an object in an unordered associative container requires both a key equality function and a hash function. The default function objects in the standard containers support a few basic types including integer types, floating point types, pointer types, and the standard strings. Since Boost.Unordered uses `boost::hash` it also supports some other types, including standard containers. To use any types not supported by these methods you have to extend Boost.Hash to support the type or use your own custom equality predicates and hash functions. See the Equality Predicates and Hash Functions section for more details.

## Basics of Hash Tables

The containers are made up of a number of *buckets*, each of which can contain any number of elements. For example, the following diagram shows a `boost::unordered_set` with 7 buckets containing 5 elements, A , B , C , D and E (this is just for illustration, containers will typically have more buckets).



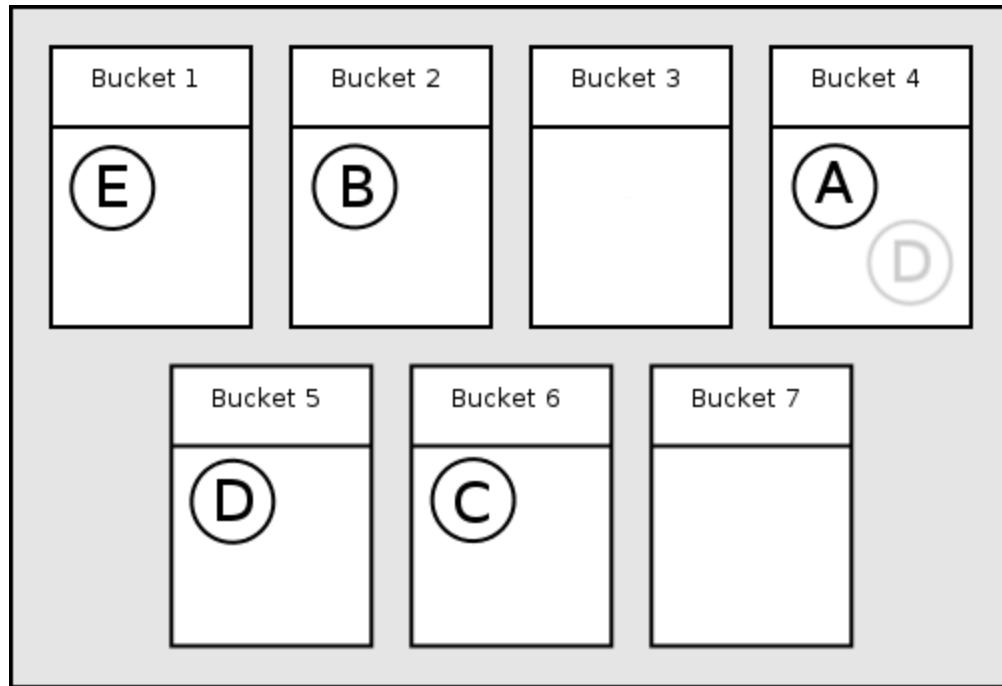
In order to decide which bucket to place an element in, the container applies the hash function, `Hash`, to the element's key (for sets the key is the whole element, but is referred to as the key so that the same terminology can be used for sets and maps). This returns a value of type `std::size_t`. `std::size_t` has a much greater range of values than the number of buckets, so the container applies another transformation to that value to choose a bucket to place the element in.

Retrieving the elements for a given key is simple. The same process is applied to the key to find the correct bucket. Then the key is compared with the elements in the bucket to find any elements that match (using the equality predicate `Pred`). If the hash function has worked well the elements will be evenly distributed amongst the buckets so only a small number of elements will need to be examined.

There is more information on hash functions and equality predicates in the next section.

You can see in the diagram that A & D have been placed in the same bucket. When looking for elements in this bucket up to 2 comparisons are made, making the search slower. This is known as a **collision**. To keep things fast we try to keep collisions to a minimum.

If instead of `boost::unordered_set` we had used `boost::unordered_flat_set`, the diagram would look as follows:



In open-addressing containers, buckets can hold at most one element; if a collision happens (like is the case of D in the example), the element uses some other available bucket in the vicinity of the original position. Given this simpler scenario, Boost.Unordered open-addressing containers offer a very limited API for accessing buckets.

*Table 2. Methods for Accessing Buckets*

All containers	
Method	Description
<code>size_type bucket_count() const</code>	The number of buckets.
<b>Closed-addressing containers only</b>	

Method	Description
<code>size_type max_bucket_count() const</code>	An upper bound on the number of buckets.
<code>size_type bucket_size(size_type n) const</code>	The number of elements in bucket n.
<code>size_type bucket(key_type const&amp; k) const</code>	Returns the index of the bucket which would contain k.
<code>local_iterator begin(size_type n)</code>	
<code>local_iterator end(size_type n)</code>	
<code>const_local_iterator begin(size_type n) const</code>	
<code>const_local_iterator end(size_type n) const</code>	Return begin and end iterators for bucket n.
<code>const_local_iterator cbegin(size_type n) const</code>	
<code>const_local_iterator cend(size_type n) const</code>	

## Controlling the Number of Buckets

As more elements are added to an unordered associative container, the number of collisions will increase causing performance to degrade. To combat this the containers increase the bucket count as elements are inserted. You can also tell the container to change the bucket count (if required) by calling `rehash`.

The standard leaves a lot of freedom to the implementer to decide how the number of buckets is chosen, but it does make some requirements based on the container's *load factor*, the number of elements divided by the number of buckets. Containers also have a *maximum load factor* which they should try to keep the load factor below.

You can't control the bucket count directly but there are two ways to influence it:

- Specify the minimum number of buckets when constructing a container or when calling `rehash`.
- Suggest a maximum load factor by calling `max_load_factor`.

`max_load_factor` doesn't let you set the maximum load factor yourself, it just lets you give a *hint*. And even then, the standard doesn't actually require the container to pay much attention to this value. The only time the load factor is *required* to be less than the maximum is following a call to `rehash`. But most implementations will try to keep the number of elements below the max load factor, and set the maximum load factor to be the same as or close to the hint - unless your hint is unreasonably small or large.

*Table 3. Methods for Controlling Bucket Size*

All containers	
Method	Description
<code>X(size_type n)</code>	Construct an empty container with at least n buckets (X is the container type).

X(InputIterator i, InputIterator j, size_type n)	Construct an empty container with at least n buckets and insert elements from the range [i, j) (X is the container type).
float load_factor() const	The average number of elements per bucket.
float max_load_factor() const	Returns the current maximum load factor.
float max_load_factor(float z)	Changes the container's maximum load factor, using z as a hint. <b>Open-addressing containers:</b> this function does nothing: users are not allowed to change the maximum load factor.
void rehash(size_type n)	Changes the number of buckets so that there are at least n buckets, and so that the load factor is less than the maximum load factor.
<b>Open-addressing and concurrent containers only</b>	
Method	Description
size_type max_load() const	Returns the maximum number of allowed elements in the container before rehash.

A note on `max_load` for open-addressing and concurrent containers: the maximum load will be (`max_load_factor()` \* `bucket_count()`) right after `rehash` or on container creation, but may slightly decrease when erasing elements in high-load situations. For instance, if we have a `boost::unordered_flat_map` with `size()` almost at `max_load()` level and then erase 1,000 elements, `max_load()` may decrease by around a few dozen elements. This is done internally by Boost.Unordered in order to keep its performance stable, and must be taken into account when planning for rehash-free insertions.

## Equality Predicates and Hash Functions

While the associative containers use an ordering relation to specify how the elements are stored, the unordered associative containers use an equality predicate and a hash function. For example, `boost::unordered_map` is declared as:

```
template <
    class Key, class Mapped,
    class Hash = boost::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<Key const, Mapped>>>
class unordered_map;
```

C++

The hash function comes first as you might want to change the hash function but not the equality predicate. For example, if you wanted to use the [FNV-1a hash](https://en.wikipedia.org/wiki/Fowler%20Noll%20Vo_hash_function#FNV-1a_hash) ([https://en.wikipedia.org/wiki/Fowler%20Noll%20Vo\\_hash\\_function#FNV-1a\\_hash](https://en.wikipedia.org/wiki/Fowler%20Noll%20Vo_hash_function#FNV-1a_hash)) you could write:

```
boost::unordered_map<std::string, int, hash::fnv_1a>
dictionary;
```

C++

There is an [implementation of FNV-1a](#) in the examples directory.

If you wish to use a different equality function, you will also need to use a matching hash function. For example, to implement a case insensitive dictionary you need to define a case insensitive equality predicate and hash function:

```
C++  
struct iequal_to  
{  
    bool operator()(std::string const& x,  
                      std::string const& y) const  
    {  
        return boost::algorithm::iequals(x, y, std::locale());  
    }  
};  
  
struct ihash  
{  
    std::size_t operator()(std::string const& x) const  
    {  
        std::size_t seed = 0;  
        std::locale locale;  
  
        for(std::string::const_iterator it = x.begin();  
             it != x.end(); ++it)  
        {  
            boost::hash_combine(seed, std::toupper(*it, locale));  
        }  
  
        return seed;  
    }  
};
```

Which you can then use in a case insensitive dictionary:

```
C++  
boost::unordered_map<std::string, int, ihash, iequal_to>  
idictionary;
```

This is a simplified version of the example at [/libs/unordered/examples/case\\_insensitive.hpp](#) which supports other locales and string types.

#### CAUTION

Be careful when using the equality (`==`) operator with custom equality predicates, especially if you're using a function pointer. If you compare two containers with different equality predicates then the result is undefined. For most stateless function objects this is impossible - since you can only compare objects with the same equality predicate you know the equality predicates must be equal. But if you're using function pointers or a stateful equality predicate (e.g. `boost::function`) then you can get into trouble.

## Custom Types

Similarly, a custom hash function can be used for custom types:

```

struct point {
    int x;
    int y;
};

bool operator==(point const& p1, point const& p2)
{
    return p1.x == p2.x && p1.y == p2.y;
}

struct point_hash
{
    std::size_t operator()(point const& p) const
    {
        std::size_t seed = 0;
        boost::hash_combine(seed, p.x);
        boost::hash_combine(seed, p.y);
        return seed;
    }
};

boost::unordered_multiset<point, point_hash> points;

```

Since the default hash function is [Boost.Hash](#), we can extend it to support the type so that the hash function doesn't need to be explicitly given:

```

struct point {
    int x;
    int y;
};

bool operator==(point const& p1, point const& p2)
{
    return p1.x == p2.x && p1.y == p2.y;
}

std::size_t hash_value(point const& p) {
    std::size_t seed = 0;
    boost::hash_combine(seed, p.x);
    boost::hash_combine(seed, p.y);
    return seed;
}

// Now the default function objects work.
boost::unordered_multiset<point> points;

```

See the [Boost.Hash documentation](#) for more detail on how to do this. Remember that it relies on extensions to the standard - so it won't work for other implementations of the unordered associative containers, you'll need to explicitly use Boost.Hash.

*Table 4 Methods for accessing the hash and equality functions*

Method	Description
hasher hash_function() <b>const</b>	Returns the container's hash function.
key_equal key_eq() <b>const</b>	Returns the container's key equality function..

# Regular Containers

Boost.Unordered closed-addressing containers (`boost::unordered_set`, `boost::unordered_map`, `boost::unordered_multiset` and `boost::unordered_multimap`) are fully conformant with the C++ specification for unordered associative containers, so for those who know how to use `std::unordered_set`, `std::unordered_map`, etc., their homonyms in Boost.Unordered are drop-in replacements. The interface of open-addressing containers (`boost::unordered_node_set`, `boost::unordered_node_map`, `boost::unordered_flat_set` and `boost::unordered_flat_map`) is very similar, but they present some minor differences listed in the dedicated standard compliance section.

For readers without previous experience with hash containers but familiar with normal associative containers (`std::set`, `std::map`, `std::multiset` and `std::multimap`), Boost.Unordered containers are used in a similar manner:

```
typedef boost::unordered_map<std::string, int> map;
map x;
x["one"] = 1;
x["two"] = 2;
x["three"] = 3;

assert(x.at("one") == 1);
assert(x.find("missing") == x.end());
```

CPP

But since the elements aren't ordered, the output of:

```
for(const map::value_type& i: x) {
    std::cout<<i.first<<", "<<i.second<<"\n";
}
```

C++

can be in any order. For example, it might be:

```
two,2
one,1
three,3
```

C++

There are other differences, which are listed in the Comparison with Associative Containers section.

## Iterator Invalidiation

It is not specified how member functions other than `rehash` and `reserve` affect the bucket count, although `insert` can only invalidate iterators when the insertion causes the container's load to be greater than the maximum allowed. For most implementations this means that `insert` will only change the number of buckets when this happens. Iterators can be invalidated by calls to `insert`, `rehash` and `reserve`.

As for pointers and references, they are never invalidated for node-based containers (`boost::unordered_[multi]set`, `boost::unordered_[multi]map`, `boost::unordered_node_set`, `boost::unordered_node_map`), but they will be when rehashing occurs for `boost::unordered_flat_set` and `boost::unordered_flat_map`: this is because these containers store elements directly into their holding buckets, so when allocating a new bucket array the elements must be transferred by means of move construction.

In a similar manner to using `reserve` for `vector`s, it can be a good idea to call `reserve` before inserting a large number of elements. This will get the expensive rehashing out of the way and let you store iterators, safe in the knowledge that they won't be invalidated. If you are inserting `n` elements into container `x`, you could first call:

```
x.reserve(n);
```

C++

## Note

`reserve(n)` reserves space for at least `n` elements, allocating enough buckets so as to not exceed the maximum load factor.

Because the maximum load factor is defined as the number of elements divided by the total number of available buckets, this function is logically equivalent to:

```
x.rehash(std::ceil(n / x.max_load_factor()))
```

C++

See the reference for more details on the `rehash` function.

## Comparison with Associative Containers

*Table 5 Interface differences*

Associative Containers	Unordered Associative Containers
Parameterized by an ordering relation <code>Compare</code>	Parameterized by a function object <code>Hash</code> and an equivalence relation <code>Pred</code>
Keys can be compared using <code>key_compare</code> which is accessed by member function <code>key_comp()</code> , values can be compared using <code>value_compare</code> which is accessed by member function <code>value_comp()</code> .	Keys can be hashed using <code>hasher</code> which is accessed by member function <code>hash_function()</code> , and checked for equality using <code>key_equal</code> which is accessed by member function <code>key_eq()</code> . There is no function object for comparing or hashing values.
Constructors have optional extra parameters for the comparison object.	Constructors have optional extra parameters for the initial minimum number of buckets, a hash function and an equality object.
Keys <code>k1</code> , <code>k2</code> are considered equivalent if <code>!Compare(k1, k2) &amp;&amp; !Compare(k2, k1)</code> .	Keys <code>k1</code> , <code>k2</code> are considered equivalent if <code>Pred(k1, k2)</code>
Member function <code>lower_bound(k)</code> and <code>upper_bound(k)</code>	No equivalent. Since the elements aren't ordered <code>lower_bound</code> and <code>upper_bound</code> would be meaningless.

Associative Containers	Unordered Associative Containers
<code>equal_range(k)</code> returns an empty range at the position that <code>k</code> would be inserted if <code>k</code> isn't present in the container.	<code>equal_range(k)</code> returns a range at the end of the container if <code>k</code> isn't present in the container. It can't return a positioned range as <code>k</code> could be inserted into multiple place. <b>Closed-addressing containers:</b> To find out the bucket that <code>k</code> would be inserted into use <code>bucket(k)</code> . But remember that an insert can cause the container to rehash - meaning that the element can be inserted into a different bucket.
<code>iterator</code> , <code>const_iterator</code> are of the bidirectional category.	<code>iterator</code> , <code>const_iterator</code> are of at least the forward category.
Iterators, pointers and references to the container's elements are never invalidated.	Iterators can be invalidated by calls to insert or rehash. <b>Node-based containers:</b> Pointers and references to the container's elements are never invalidated. <b>Flat containers:</b> Pointers and references to the container's elements are invalidated when rehashing occurs.
Iterators iterate through the container in the order defined by the comparison object.	Iterators iterate through the container in an arbitrary order, that can change as elements are inserted, although equivalent elements are always adjacent.
No equivalent	<b>Closed-addressing containers:</b> Local iterators can be used to iterate through individual buckets. (The order of local iterators and iterators aren't required to have any correspondence.)
Can be compared using the <code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> operators.	Can be compared using the <code>==</code> and <code>!=</code> operators.
	When inserting with a hint, implementations are permitted to ignore the hint.

Table 6 Complexity Guarantees

Operation	Associative Containers	Unordered Associative Containers
Construction of empty container	constant	$O(n)$ where $n$ is the minimum number of buckets.
Construction of container from a range of $N$ elements	$O(N \log N)$ , $O(N)$ if the range is sorted with <code>value_comp()</code>	Average case $O(N)$ , worst case $O(N^2)$

Operation	Associative Containers	Unordered Associative Containers
Insert a single element	logarithmic	Average case constant, worst case linear
Insert a single element with a hint	Amortized constant if t elements inserted right after hint, logarithmic otherwise	Average case constant, worst case linear (ie. the same as a normal insert).
Inserting a range of $N$ elements	$N \log(\text{size}()) + N$	Average case $O(N)$ , worst case $O(N * \text{size}())$
Erase by key, k	$O(\log(\text{size}()) + \text{count}(k))$	Average case: $O(\text{count}(k))$ , Worst case: $O(\text{size}())$
Erase a single element by iterator	Amortized constant	Average case: $O(1)$ , Worst case: $O(\text{size}())$
Erase a range of $N$ elements	$O(\log(\text{size}()) + N)$	Average case: $O(N)$ , Worst case: $O(\text{size}())$
Clearing the container	$O(\text{size}())$	$O(\text{size}())$
Find	logarithmic	Average case: $O(1)$ , Worst case: $O(\text{size}())$
Count	$O(\log(\text{size}()) + \text{count}(k))$	Average case: $O(1)$ , Worst case: $O(\text{size}())$
<code>equal_range(k)</code>	logarithmic	Average case: $O(\text{count}(k))$ , Worst case: $O(\text{size}())$
<code>lower_bound</code> , <code>upper_bound</code>	logarithmic	n/a

## Concurrent Containers

Boost.Unordered provides `boost::concurrent_flat_set` and `boost::concurrent_flat_map`, hash tables that allow concurrent write/read access from different threads without having to implement any synchronization mechanism on the user's side.

```

std::vector<int> input;
boost::concurrent_flat_map<int,int> m;

...

// process input in parallel
const int num_threads = 8;
std::vector<std::jthread> threads;
std::size_t chunk = input.size() / num_threads; // how many elements per thread

for (int i = 0; i < num_threads; ++i) {
    threads.emplace_back([&,i] {
        // calculate the portion of input this thread takes care of
        std::size_t start = i * chunk;
        std::size_t end = (i == num_threads - 1)? input.size(): (i + 1) * chunk;

        for (std::size_t n = start; n < end; ++n) {
            m.emplace(input[n], calculation(input[n]));
        }
    });
}

```

In the example above, threads access `m` without synchronization, just as we'd do in a single-threaded scenario. In an ideal setting, if a given workload is distributed among  $N$  threads, execution is  $N$  times faster than with one thread —this limit is never attained in practice due to synchronization overheads and *contention* (one thread waiting for another to leave a locked portion of the map), but Boost.Unordered concurrent containers are designed to perform with very little overhead and typically achieve *linear scaling* (that is, performance is proportional to the number of threads up to the number of logical cores in the CPU).

## Visitation-based API

The first thing a new user of `boost::concurrent_flat_set` or `boost::concurrent_flat_map` will notice is that these classes *do not provide iterators* (which makes them technically not Containers ([https://en.cppreference.com/w/cpp/named\\_req/Container](https://en.cppreference.com/w/cpp/named_req/Container)) in the C++ standard sense). The reason for this is that iterators are inherently thread-unsafe. Consider this hypothetical code:

```

auto it = m.find(k); // A: get an iterator pointing to the element with key k
if (it != m.end() ) {
    some_function(*it); // B: use the value of the element
}

```

In a multithreaded scenario, the iterator `it` may be invalid at point B if some other thread issues an `m.erase(k)` operation between A and B. There are designs that can remedy this by making iterators lock the element they point to, but this approach lends itself to high contention and can easily produce deadlocks in a program. `operator[]` has similar concurrency issues, and is not provided by `boost::concurrent_flat_map` either. Instead, element access is done through so-called *visitation functions*:

```

m.visit(k, [](const auto& x) { // x is the element with key k (if it exists)
    some_function(x);           // use it
});

```

The visitation function passed by the user (in this case, a lambda function) is executed internally by Boost.Unordered in a thread-safe manner, so it can access the element without worrying about other threads interfering in the process.

On the other hand, a visitation function can *not* access the container itself:

```
m.visit(k, [&](const auto& x) {
    some_function(x, m.size()); // forbidden: m can't be accessed inside visitation
});
```

C++

Access to a different container is allowed, though:

```
m.visit(k, [&](const auto& x) {
    if (some_function(x)) {
        m2.insert(x); // OK, m2 is a different boost::concurrent_flat_map
    }
});
```

C++

But, in general, visitation functions should be as lightweight as possible to reduce contention and increase parallelization. In some cases, moving heavy work outside of visitation may be beneficial:

```
std::optional<value_type> o;
bool found = m.visit(k, [&](const auto& x) {
    o = x;
});
if (found) {
    some_heavy_duty_function(*o);
}
```

C++

Visitation is prominent in the API provided by `boost::concurrent_flat_set` and `boost::concurrent_flat_map`, and many classical operations have visitation-enabled variations:

```
m.insert_or_visit(x, [](auto& y) {
    // if insertion failed because of an equivalent element y,
    // do something with it, for instance:
    ++y.second; // increment the mapped part of the element
});
```

C++

Note that in this last example the visitation function could actually *modify* the element: as a general rule, operations on a `boost::concurrent_flat_map` `m` will grant visitation functions const/non-const access to the element depending on whether `m` is const/non-const. Const access can be always be explicitly requested by using `cvisit` overloads (for instance, `insert_or_cvisit`) and may result in higher parallelization. For `boost::concurrent_flat_set`, on the other hand, visitation is always const access. Consult the references of `boost::concurrent_flat_set` and `boost::concurrent_flat_map` for the complete list of visitation-enabled operations.

## Whole-Table Visitation

In the absence of iterators, `visit_all` is provided as an alternative way to process all the elements in the container:

```
m.visit_all([](auto& x) {
    x.second = 0; // reset the mapped part of the element
});
```

C++

In C++17 compilers implementing standard parallel algorithms, whole-table visitation can be parallelized:

```
m.visit_all(std::execution::par, [](auto& x) { // run in parallel
    x.second = 0; // reset the mapped part of the element
});
```

Traversal can be interrupted midway:

```
// finds the key to a given (unique) value

int key = 0;
int value = ...;
bool found = !m.visit_while([&](const auto& x) {
    if(x.second == value) {
        key = x.first;
        return false; // finish
    }
    else {
        return true; // keep on visiting
    }
});

if(found) { ... }
```

There is one last whole-table visitation operation, `erase_if`:

```
m.erase_if([](auto& x) {
    return x.second == 0; // erase the elements whose mapped value is zero
});
```

`visit_while` and `erase_if` can also be parallelized. Note that, in order to increase efficiency, whole-table visitation operations do not block the table during execution: this implies that elements may be inserted, modified or erased by other threads during visitation. It is advisable not to assume too much about the exact global state of a concurrent container at any point in your program.

## Bulk visitation

Suppose you have an `std::array` of keys you want to look up for in a concurrent map:

```
std::array<int, N> keys;
...
for(const auto& key: keys) {
    m.visit(key, [](auto& x) { ++x.second; });
}
```

*Bulk visitation* allows us to pass all the keys in one operation:

```
m.visit(keys.begin(), keys.end(), [](auto& x) { ++x.second; });
```

This functionality is not provided for mere syntactic convenience, though: by processing all the keys at once, some internal optimizations can be applied that increase performance over the regular, one-at-a-time case (consult the benchmarks). In fact, it may be beneficial to buffer incoming keys so that they can be bulk visited in chunks:

```

static constexpr auto bulk_visit_size = boost::concurrent_flat_map<int,int>::bulk_visit_size;
std::array<int, bulk_visit_size> buffer;
std::size_t i=0;
while(...) { // processing loop
    ...
    buffer[i++] = k;
    if(i == bulk_visit_size) {
        map.visit(buffer.begin(), buffer.end(), [](auto& x) { ++x.second; });
        i = 0;
    }
    ...
}
// flush remaining keys
map.visit(buffer.begin(), buffer.begin() + i, [](auto& x) { ++x.second; });

```

There's a latency/throughput tradeoff here: it will take longer for incoming keys to be processed (since they are buffered), but the number of processed keys per second is higher. `bulk_visit_size` is the recommended chunk size—smaller buffers may yield worse performance.

## Blocking Operations

`boost::concurrent_flat_set`s and `boost::concurrent_flat_map`s can be copied, assigned, cleared and merged just like any Boost.Unordered container. Unlike most other operations, these are *blocking*, that is, all other threads are prevented from accessing the tables involved while a copy, assignment, clear or merge operation is in progress. Blocking is taken care of automatically by the library and the user need not take any special precaution, but overall performance may be affected.

Another blocking operation is *rehashing*, which happens explicitly via `rehash` / `reserve` or during insertion when the table's load hits `max_load()`. As with non-concurrent containers, reserving space in advance of bulk insertions will generally speed up the process.

## Interoperability with non-concurrent containers

As open-addressing and concurrent containers are based on the same internal data structure,

`boost::unordered_flat_set` and `boost::unordered_flat_map` can be efficiently move-constructed from `boost::concurrent_flat_set` and `boost::concurrent_flat_map`, respectively, and vice versa. This interoperability comes handy in multistage scenarios where parts of the data processing happen in parallel whereas other steps are non-concurrent (or non-modifying). In the following example, we want to construct a histogram from a huge input vector of words: the population phase can be done in parallel with `boost::concurrent_flat_map` and results then transferred to the final container.

```

std::vector<std::string> words = ...;

// Insert words in parallel
boost::concurrent_flat_map<std::string_view, std::size_t> m0;
std::for_each(
    std::execution::par, words.begin(), words.end(),
    [&](const auto& word) {
        m0.try_emplace_or_visit(word, 1, [](auto& x) { ++x.second; });
    });
// Transfer to a regular unordered_flat_map
boost::unordered_flat_map m=std::move(m0);

```

# Standard Compliance

## Closed-addressing Containers

`boost::unordered_[multi]set` and `boost::unordered_[multi]map` provide a conformant implementation for C++11 (or later) compilers of the latest standard revision of C++ unordered associative containers, with very minor deviations as noted. The containers are fully [AllocatorAware](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer) ([https://en.cppreference.com/w/cpp/named\\_req/AllocatorAwareContainer](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer)) and support [fancy pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers) ([https://en.cppreference.com/w/cpp/named\\_req/Allocator#Fancy\\_pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers)).

## Deduction Guides

Deduction guides for [class template argument deduction \(CTAD\)](#)

([https://en.cppreference.com/w/cpp/language/class\\_template\\_argument\\_deduction](https://en.cppreference.com/w/cpp/language/class_template_argument_deduction)) are only available on C++17 (or later) compilers.

## Piecewise Pair Emplacement

In accordance with the standard specification, `boost::unordered_[multi]map::emplace` supports piecewise pair construction:

```
boost::unordered_multimap<std::string, std::complex> x;  
  
x.emplace(  
    std::piecewise_construct,  
    std::make_tuple("key"), std::make_tuple(1, 2));
```

C++

Additionally, the same functionality is provided via non-standard `boost::unordered::piecewise_construct` and `Boost.Tuple`:

```
x.emplace(  
    boost::unordered::piecewise_construct,  
    boost::make_tuple("key"), boost::make_tuple(1, 2));
```

C++

This feature has been retained for backwards compatibility with previous versions of Boost.Unordered: users are encouraged to update their code to use `std::piecewise_construct` and `std::tuple`s instead.

## Swap

When swapping, `Pred` and `Hash` are not currently swapped by calling `swap`, their copy constructors are used. As a consequence, when swapping an exception may be thrown from their copy constructor.

## Open-addressing Containers

The C++ standard does not currently provide any open-addressing container specification to adhere to, so

`boost::unordered_flat_set` / `unordered_node_set` and `boost::unordered_flat_map` / `unordered_node_map` take inspiration from `std::unordered_set` and `std::unordered_map`, respectively, and depart from their interface where convenient or as dictated by their internal data structure, which is radically different from that imposed by the standard (closed addressing).

Open-addressing containers provided by Boost.Unordered only work with reasonably compliant C++11 (or later) compilers. Language-level features such as move semantics and variadic template parameters are then not emulated. The containers are fully [AllocatorAware](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer) ([https://en.cppreference.com/w/cpp/named\\_req/AllocatorAwareContainer](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer)) and support [fancy pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers) ([https://en.cppreference.com/w/cpp/named\\_req/Allocator#Fancy\\_pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers)).

The main differences with C++ unordered associative containers are:

- In general:
  - `begin()` is not constant-time.
  - `erase(iterator)` does not return an iterator to the following element, but a proxy object that converts to that iterator if requested; this avoids a potentially costly iterator increment operation when not needed.
  - There is no API for bucket handling (except `bucket_count`).
  - The maximum load factor of the container is managed internally and can't be set by the user. The maximum load, exposed through the public function `max_load`, may decrease on erasure under high-load conditions.
- Flat containers (`boost::unordered_flat_set` and `boost::unordered_flat_map`):
  - `value_type` must be move-constructible.
  - Pointer stability is not kept under rehashing.
  - There is no API for node extraction/insertion.

## Concurrent Containers

There is currently no specification in the C++ standard for this or any other type of concurrent data structure. The APIs of `boost::concurrent_flat_set` and `boost::concurrent_flat_map` are modelled after `std::unordered_flat_set` and `std::unordered_flat_map`, respectively, with the crucial difference that iterators are not provided due to their inherent problems in concurrent scenarios (high contention, prone to deadlocking): so, Boost.Unordered concurrent containers are technically not models of [Container](https://en.cppreference.com/w/cpp/named_req/Container) ([https://en.cppreference.com/w/cpp/named\\_req/Container](https://en.cppreference.com/w/cpp/named_req/Container)), although they meet all the requirements of [AllocatorAware](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer) ([https://en.cppreference.com/w/cpp/named\\_req/AllocatorAwareContainer](https://en.cppreference.com/w/cpp/named_req/AllocatorAwareContainer)) containers (including [fancy pointer](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers) ([https://en.cppreference.com/w/cpp/named\\_req/Allocator#Fancy\\_pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers)) support) except those implying iterators.

In a non-concurrent unordered container, iterators serve two main purposes:

- Access to an element previously located via lookup.
- Container traversal.

In place of iterators, `boost::concurrent_flat_set` and `boost::concurrent_flat_map` use *internal visitation* facilities as a thread-safe substitute. Classical operations returning an iterator to an element already existing in the container, like for instance:

```
iterator find(const key_type& k);
std::pair<iterator, bool> insert(const value_type& obj);
```

C++

are transformed to accept a *visitation function* that is passed such element:

```
template<class F> size_t visit(const key_type& k, F f);
template<class F> bool insert_or_visit(const value_type& obj, F f);
```

C++

(In the second case `f` is only invoked if there's an equivalent element to `obj` in the table, not if insertion is successful). Container traversal is served by:

```
template<class F> size_t visit_all(F f);
```

of which there are parallelized versions in C++17 compilers with parallel algorithm support. In general, the interface of concurrent containers is derived from that of their non-concurrent counterparts by a fairly straightforward process of replacing iterators with visitation where applicable. If for regular maps `iterator` and `const_iterator` provide mutable and const access to elements, respectively, here visitation is granted mutable or const access depending on the constness of the member function used (there are also `*cvisit` overloads for explicit const visitation); In the case of `boost::concurrent_flat_set`, visitation is always const.

One notable operation not provided by `boost::concurrent_flat_map` is `operator[] / at`, which can be replaced, if in a more convoluted manner, by `try_emplace_or_visit`.

## Data Structures

### Closed-addressing Containers

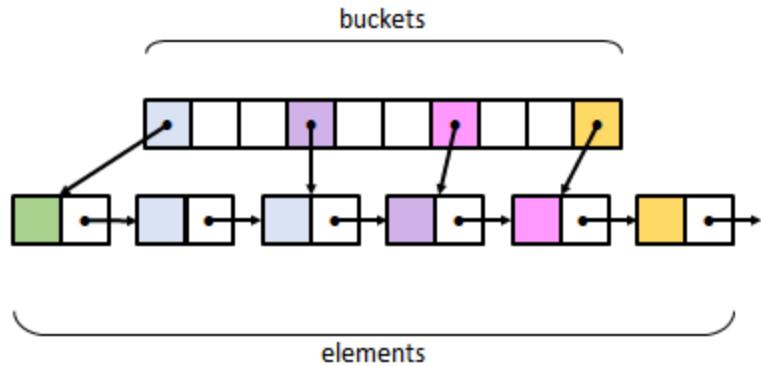
Boost.Unordered sports one of the fastest implementations of closed addressing, also commonly known as separate chaining ([https://en.wikipedia.org/wiki/Hash\\_table#Separate\\_chaining](https://en.wikipedia.org/wiki/Hash_table#Separate_chaining)). An example figure representing the data structure is below:



*Figure 1. A simple bucket group approach*

An array of "buckets" is allocated and each bucket in turn points to its own individual linked list. This makes meeting the standard requirements of bucket iteration straight-forward. Unfortunately, iteration of the entire container is often times slow using this layout as each bucket must be examined for occupancy, yielding a time complexity of `O(bucket_count() + size())` when the standard requires complexity to be `O(size())`.

Canonical standard implementations will wind up looking like the diagram below:



*Figure 2. The canonical standard approach*

It's worth noting that this approach is only used by libc++ and libstdc++; the MSVC Dinkumware implementation uses a different one. A more detailed analysis of the standard containers can be found [here](http://bannalia.blogspot.com/2013/10/implementation-of-c-unordered.html) (<http://bannalia.blogspot.com/2013/10/implementation-of-c-unordered.html>).

This unusually laid out data structure is chosen to make iteration of the entire container efficient by inter-connecting all of the nodes into a singly-linked list. One might also notice that buckets point to the node *before* the start of the bucket's elements. This is done so that removing elements from the list can be done efficiently without introducing the need for a doubly-linked list. Unfortunately, this data structure introduces a guaranteed extra indirection. For example, to access the first element of a bucket, something like this must be done:

```
auto const idx = get_bucket_idx(hash_function(key));
node* p = buckets[idx]; // first load
node* n = p->next; // second load
if (n && is_in_bucket(n, idx)) {
    value_type const& v = *n; // third load
    // ...
}
```

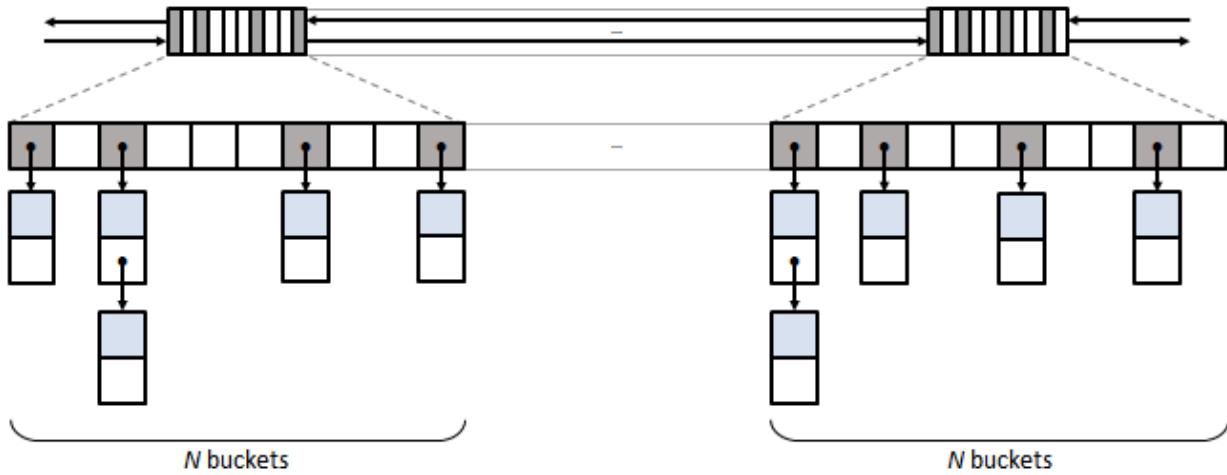
C++

With a simple bucket group layout, this is all that must be done:

```
auto const idx = get_bucket_idx(hash_function(key));
node* n = buckets[idx]; // first load
if (n) {
    value_type const& v = *n; // second load
    // ...
}
```

C++

In practice, the extra indirection can have a dramatic performance impact to common operations such as `insert`, `find` and `erase`. But to keep iteration of the container fast, Boost.Unordered introduces a novel data structure, a "bucket group". A bucket group is a fixed-width view of a subsection of the buckets array. It contains a bitmask (a `std::size_t`) which it uses to track occupancy of buckets and contains two pointers so that it can form a doubly-linked list with non-empty groups. An example diagram is below:



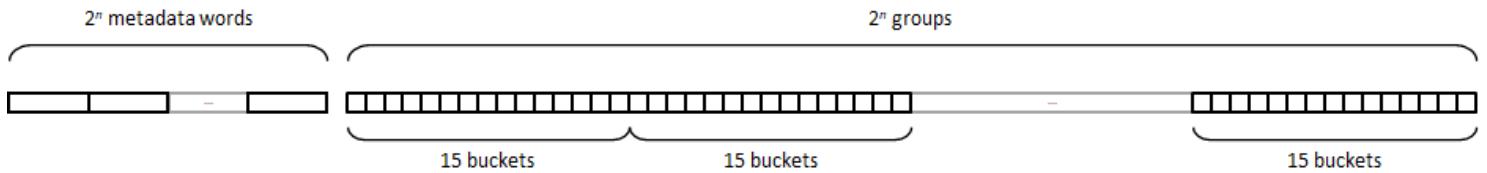
*Figure 3. The new layout used by Boost*

Thus container-wide iteration is turned into traversing the non-empty bucket groups (an operation with constant time complexity) which reduces the time complexity back to  $O(\text{size}())$ . In total, a bucket group is only 4 words in size and it views `sizeof(std::size_t) * CHAR_BIT` buckets meaning that for all common implementations, there's only 4 bits of space overhead per bucket introduced by the bucket groups.

A more detailed description of Boost.Unordered's closed-addressing implementation is given in an [external article](https://bannalia.blogspot.com/2022/06/advancing-state-of-art-for.html) (<https://bannalia.blogspot.com/2022/06/advancing-state-of-art-for.html>). For more information on implementation rationale, read the corresponding section.

## Open-addressing Containers

The diagram shows the basic internal layout of `boost::unordered_flat_set / unordered_node_set` and `boost::unordered_flat_map / unordered_node_map`.



*Figure 4. Open-addressing layout used by Boost.Unordered.*

As with all open-addressing containers, elements (or pointers to the element nodes in the case of `boost::unordered_node_set` and `boost::unordered_node_map`) are stored directly in the bucket array. This array is logically divided into  $2^n$  groups of 15 elements each. In addition to the bucket array, there is an associated *metadata array* with  $2^n$  16-byte words.

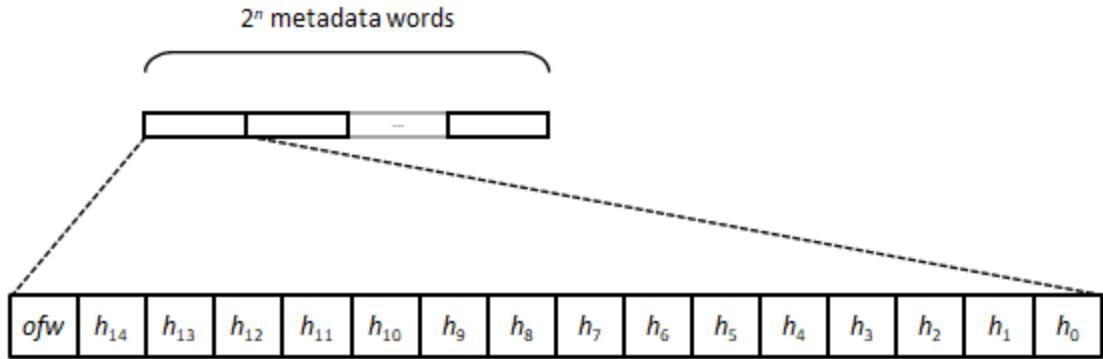


Figure 5. Breakdown of a metadata word.

A metadata word is divided into  $15 h_i$  bytes (one for each associated bucket), and an *overflow byte* (*ofw* in the diagram). The value of  $h_i$  is:

- 0 if the corresponding bucket is empty.
- 1 to encode a special empty bucket called a *sentinel*, which is used internally to stop iteration when the container has been fully traversed.
- If the bucket is occupied, a *reduced hash value* obtained from the hash value of the element.

When looking for an element with hash value  $h$ , SIMD technologies such as [SSE2](https://en.wikipedia.org/wiki/SSE2) (<https://en.wikipedia.org/wiki/SSE2>) and [Neon](https://en.wikipedia.org/wiki/ARM_architecture_family#Advanced SIMD_(Neon)) ([https://en.wikipedia.org/wiki/ARM\\_architecture\\_family#Advanced SIMD\\_\(Neon\)](https://en.wikipedia.org/wiki/ARM_architecture_family#Advanced SIMD_(Neon))) allow us to very quickly inspect the full metadata word and look for the reduced value of  $h$  among all the 15 buckets with just a handful of CPU instructions: non-matching buckets can be readily discarded, and those whose reduced hash value matches need be inspected via full comparison with the corresponding element. If the looked-for element is not present, the overflow byte is inspected:

- If the bit in the position  $h \bmod 8$  is zero, lookup terminates (and the element is not present).
- If the bit is set to 1 (the group has been *overflown*), further groups are checked using [quadratic probing](https://en.wikipedia.org/wiki/Quadratic_probing) ([https://en.wikipedia.org/wiki/Quadratic\\_probing](https://en.wikipedia.org/wiki/Quadratic_probing)), and the process is repeated.

Insertion is algorithmically similar: empty buckets are located using SIMD, and when going past a full group its corresponding overflow bit is set to 1.

In architectures without SIMD support, the logical layout stays the same, but the metadata word is codified using a technique we call *bit interleaving*: this layout allows us to emulate SIMD with reasonably good performance using only standard arithmetic and logical operations.

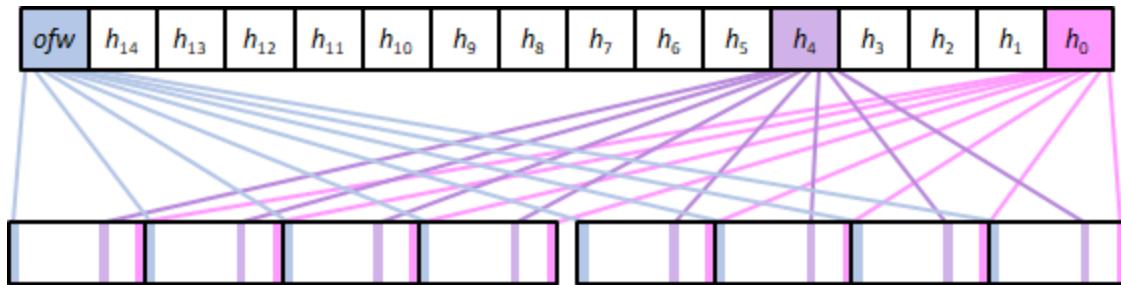


Figure 6. Bit-interleaved metadata word.

A more detailed description of Boost.Unordered's open-addressing implementation is given in an [external article](#) (<https://bannalia.blogspot.com/2022/11/inside-boostunorderedflatmap.html>). For more information on implementation rationale, read the corresponding section.

## Concurrent Containers

`boost::concurrent_flat_set` and `boost::concurrent_flat_map` use the basic open-addressing layout described above augmented with synchronization mechanisms.

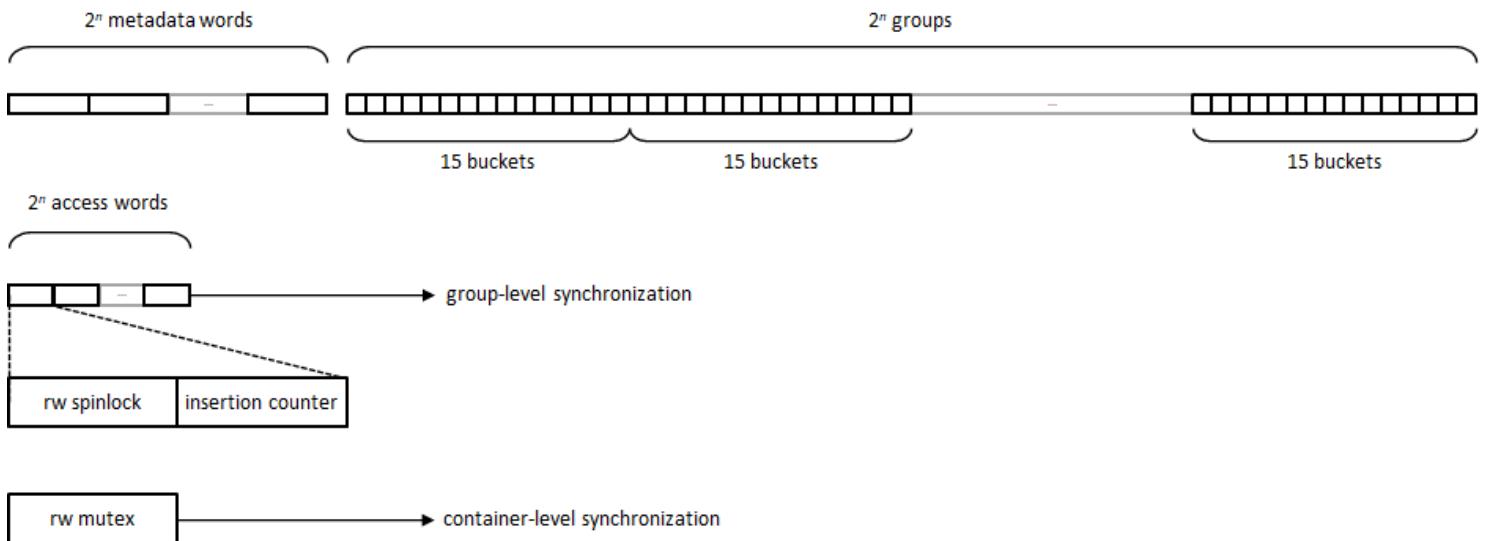


Figure 7. Concurrent open-addressing layout used by Boost.Unordered.

Two levels of synchronization are used:

- Container level: A read-write mutex is used to control access from any operation to the container. Typically, such access is in read mode (that is, concurrent) even for modifying operations, so for most practical purposes there is no thread contention at this level. Access is only in write mode (blocking) when rehashing or performing container-wide operations such as swapping or assignment.
- Group level: Each 15-slot group is equipped with an 8-byte word containing:
  - A read-write spinlock for synchronized access to any element in the group.
  - An atomic *insertion counter* used for optimistic insertion as described below.

By using atomic operations to access the group metadata, lookup is (group-level) lock-free up to the point where an actual comparison needs to be done with an element that has been previously SIMD-matched: only then is the group's spinlock used.

Insertion uses the following *optimistic algorithm*:

- The value of the insertion counter for the initial group in the probe sequence is locally recorded (let's call this value  $c_0$ ).
- Lookup is as described above. If lookup finds no equivalent element, search for an available slot for insertion successively locks/unlocks each group in the probing sequence.

- When an available slot is located, it is preemptively occupied (its reduced hash value is set) and the insertion counter is atomically incremented: if no other thread has incremented the counter during the whole operation (which is checked by comparing with `c0`), then we're good to go and complete the insertion, otherwise we roll back and start over.
- This algorithm has very low contention both at the lookup and actual insertion phases in exchange for the possibility that computations have to be started over if some other thread interferes in the process by performing a successful insertion beginning at the same group. In practice, the start-over frequency is extremely small, measured in the range of parts per million for some of our benchmarks.

For more information on implementation rationale, read the corresponding section.

## Benchmarks

### `boost::unordered_[multi]set`

All benchmarks were created using `unordered_set<unsigned int>` (non-duplicate) and `unordered_multiset<unsigned int>` (duplicate). The source code can be [found here](#)

([https://github.com/boostorg/boost\\_unordered\\_benchmarks/tree/boost\\_unordered\\_set](https://github.com/boostorg/boost_unordered_benchmarks/tree/boost_unordered_set)).

The insertion benchmarks insert `n` random values, where `n` is between 10,000 and 3 million. For the duplicated benchmarks, the same random values are repeated an average of 5 times.

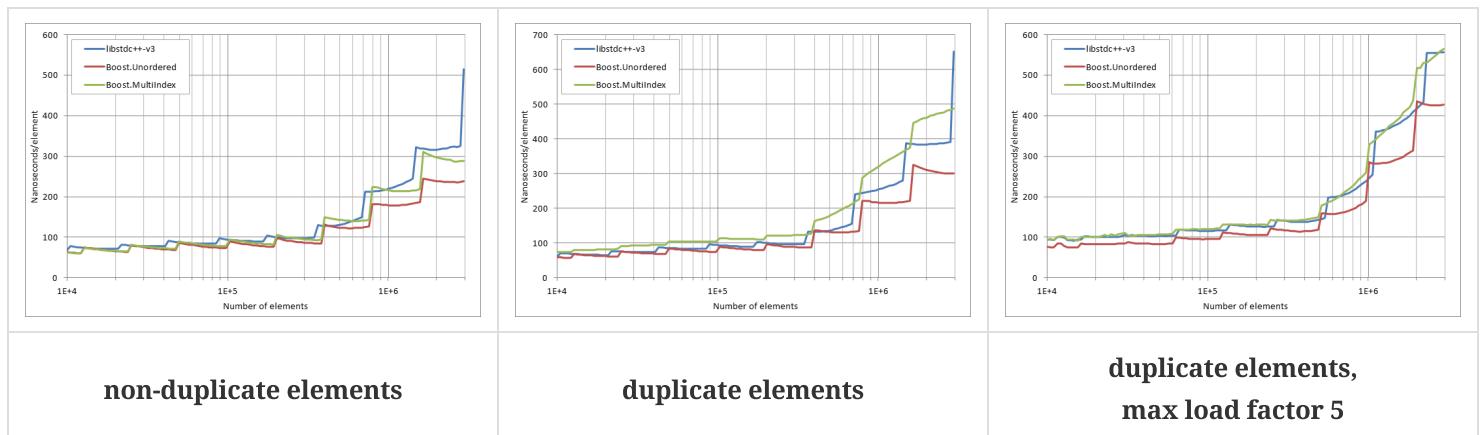
The erasure benchmarks erase all `n` elements randomly until the container is empty. Erasure by key uses `erase(const key_type&)` to remove entire groups of equivalent elements in each operation.

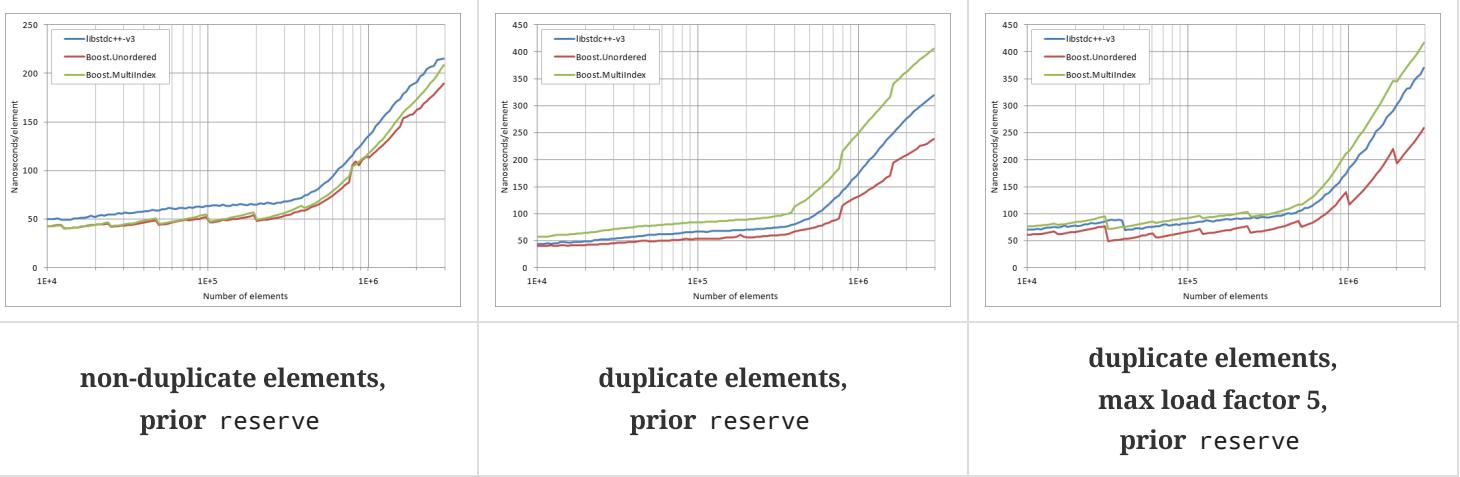
The successful lookup benchmarks are done by looking up all `n` values, in their original insertion order.

The unsuccessful lookup benchmarks use `n` randomly generated integers but using a different seed value.

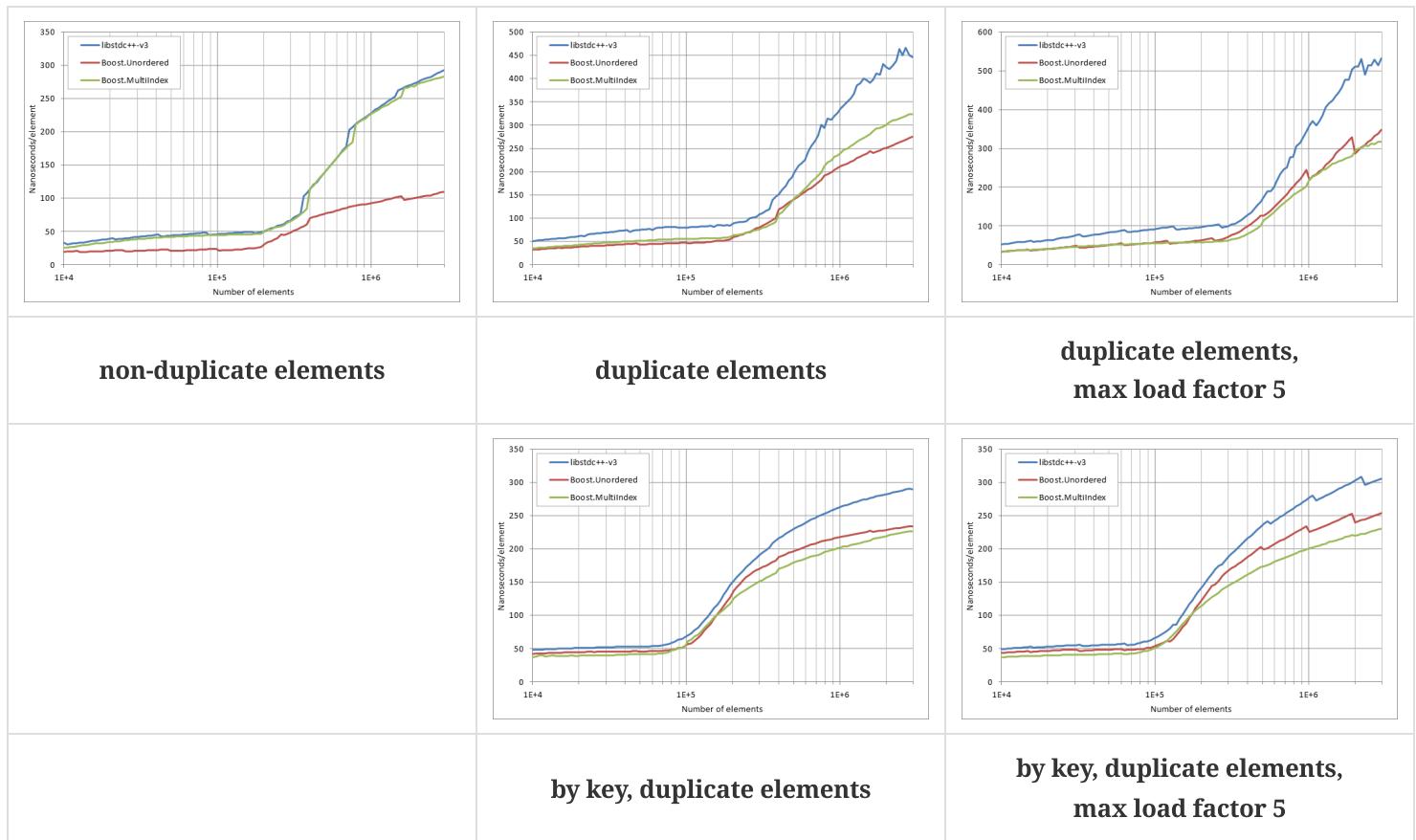
### GCC 12 + libstdc++-v3, x64

#### Insertion

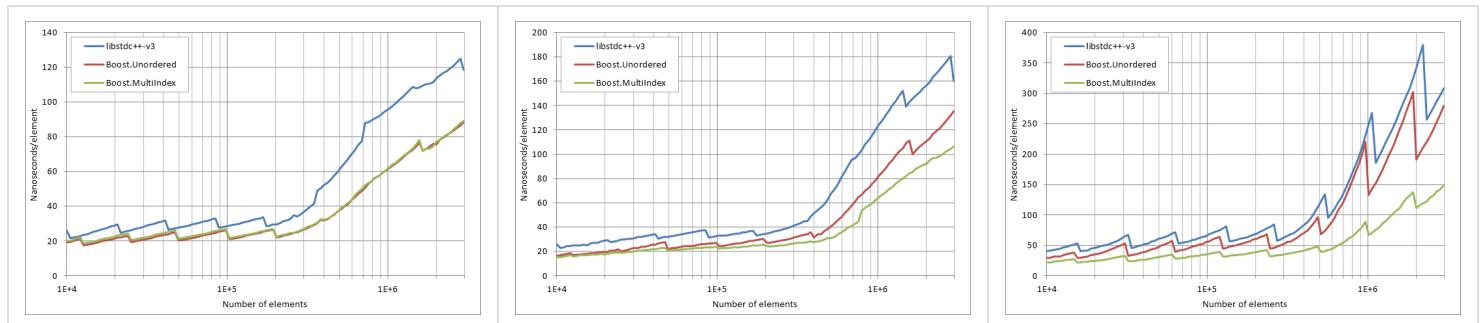




## Erasure



## Successful Lookup

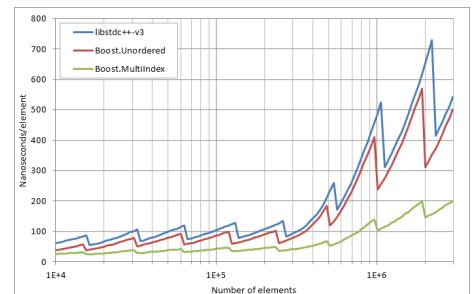
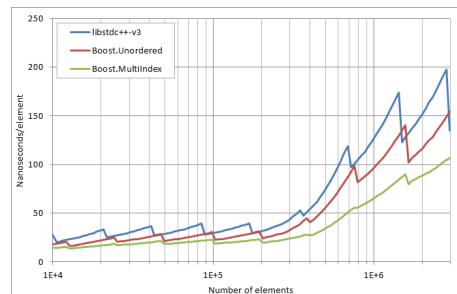
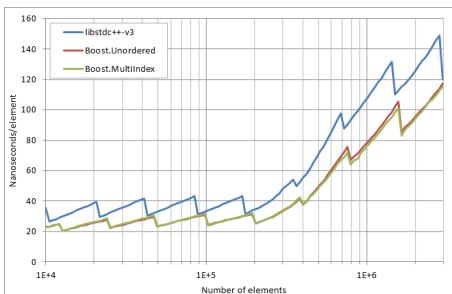


**non-duplicate elements**

**duplicate elements**

**duplicate elements,  
max load factor 5**

Unsuccessful lookup



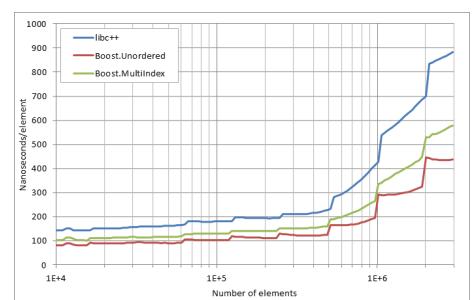
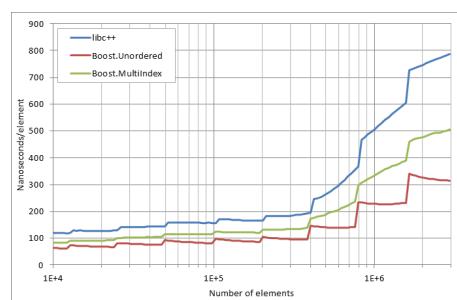
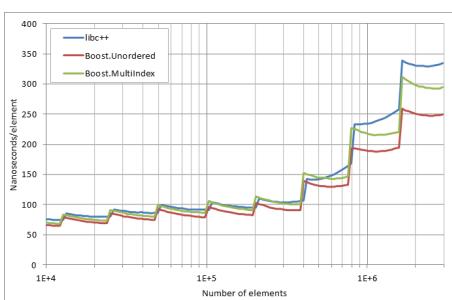
**non-duplicate elements**

**duplicate elements**

**duplicate elements,  
max load factor 5**

Clang 15 + libc++, x64

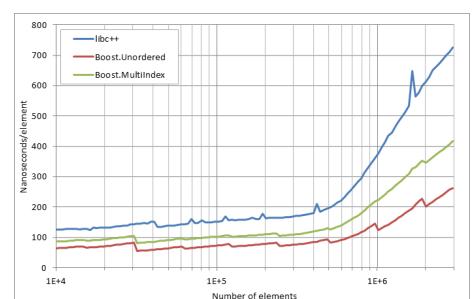
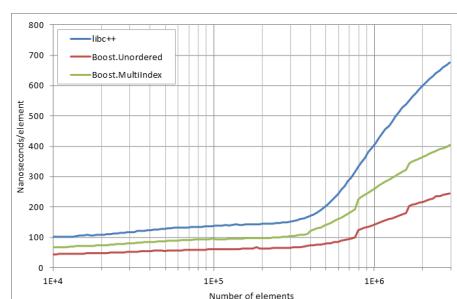
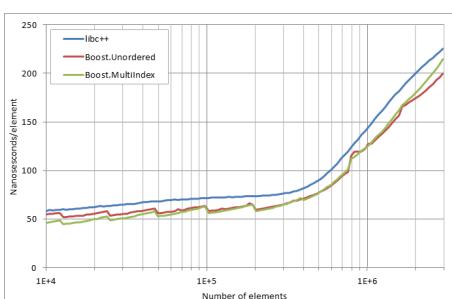
Insertion



**non-duplicate elements**

**duplicate elements**

**duplicate elements,  
max load factor 5**

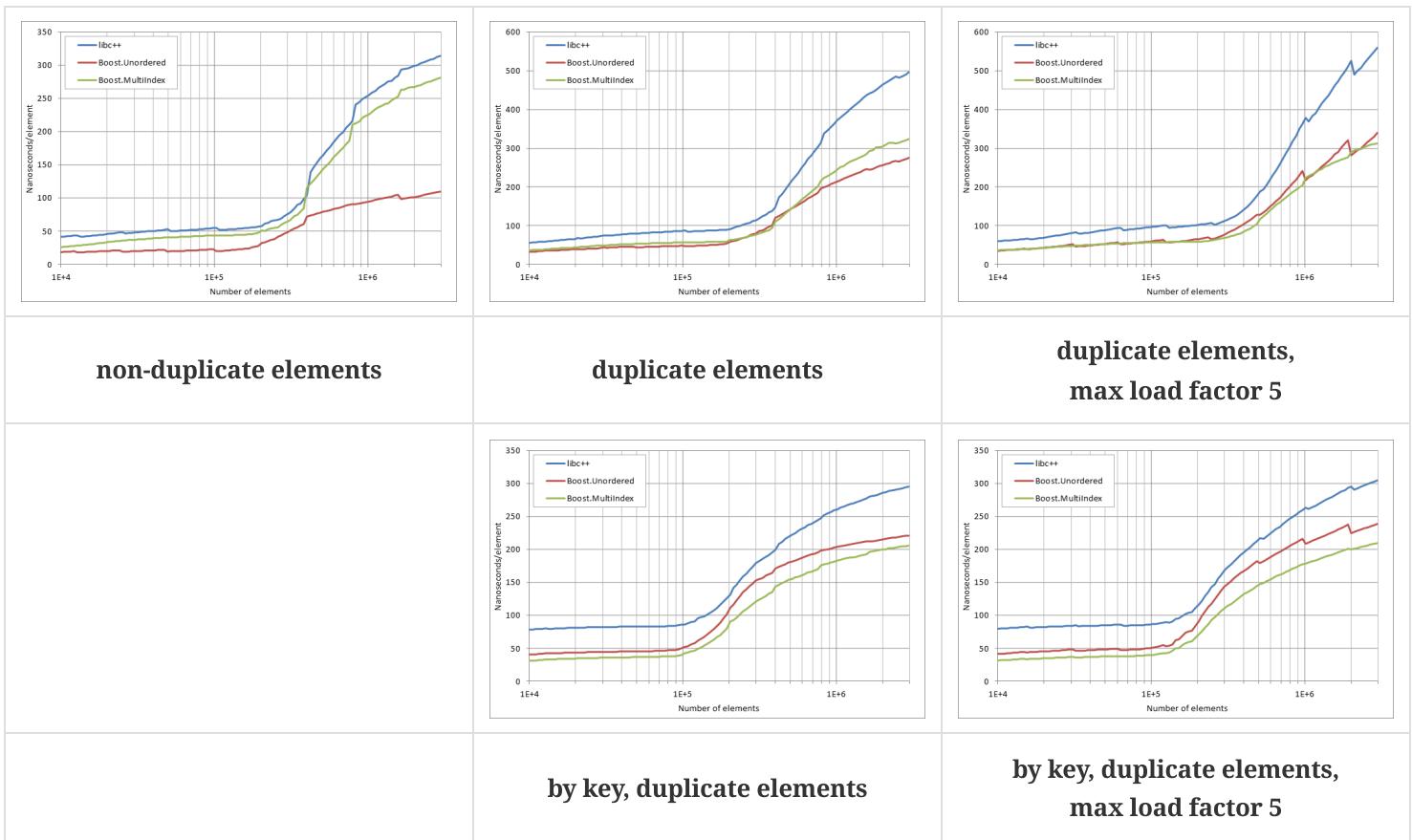


**non-duplicate elements,  
prior reserve**

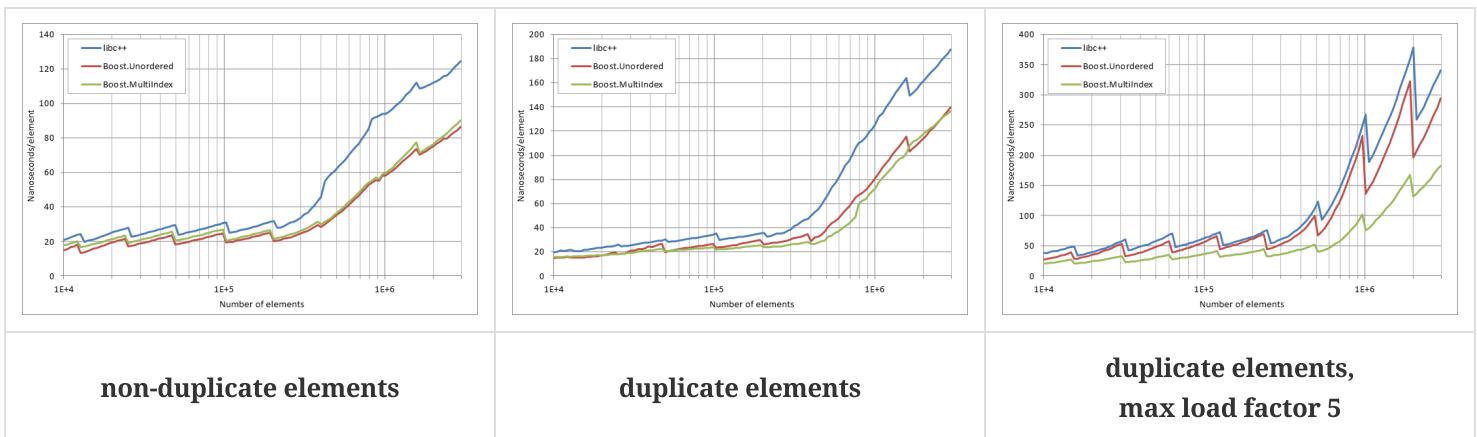
**duplicate elements,  
prior reserve**

**duplicate elements,  
max load factor 5,  
prior reserve**

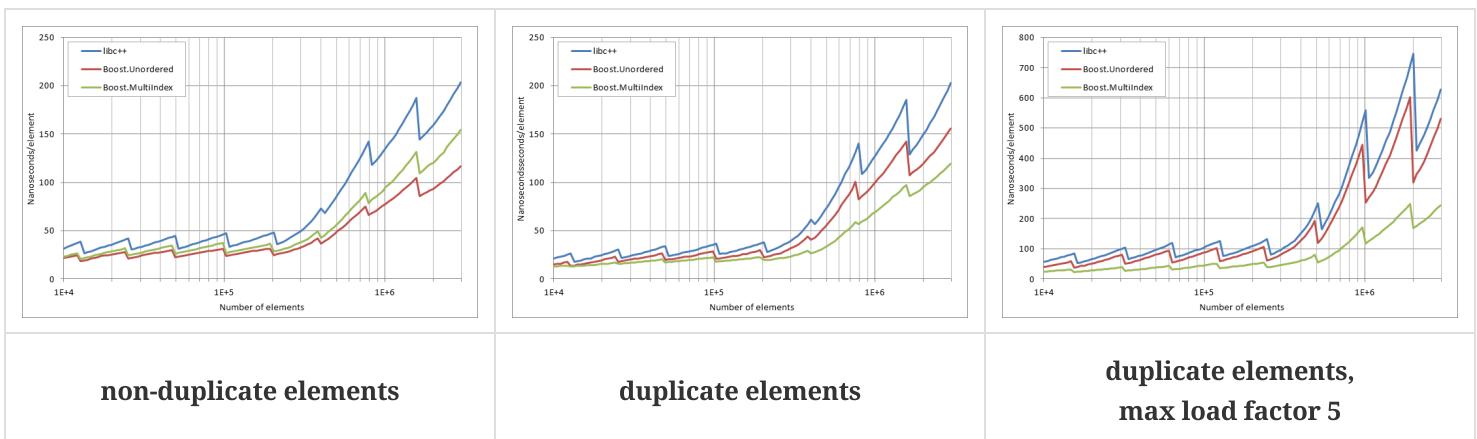
Erasure



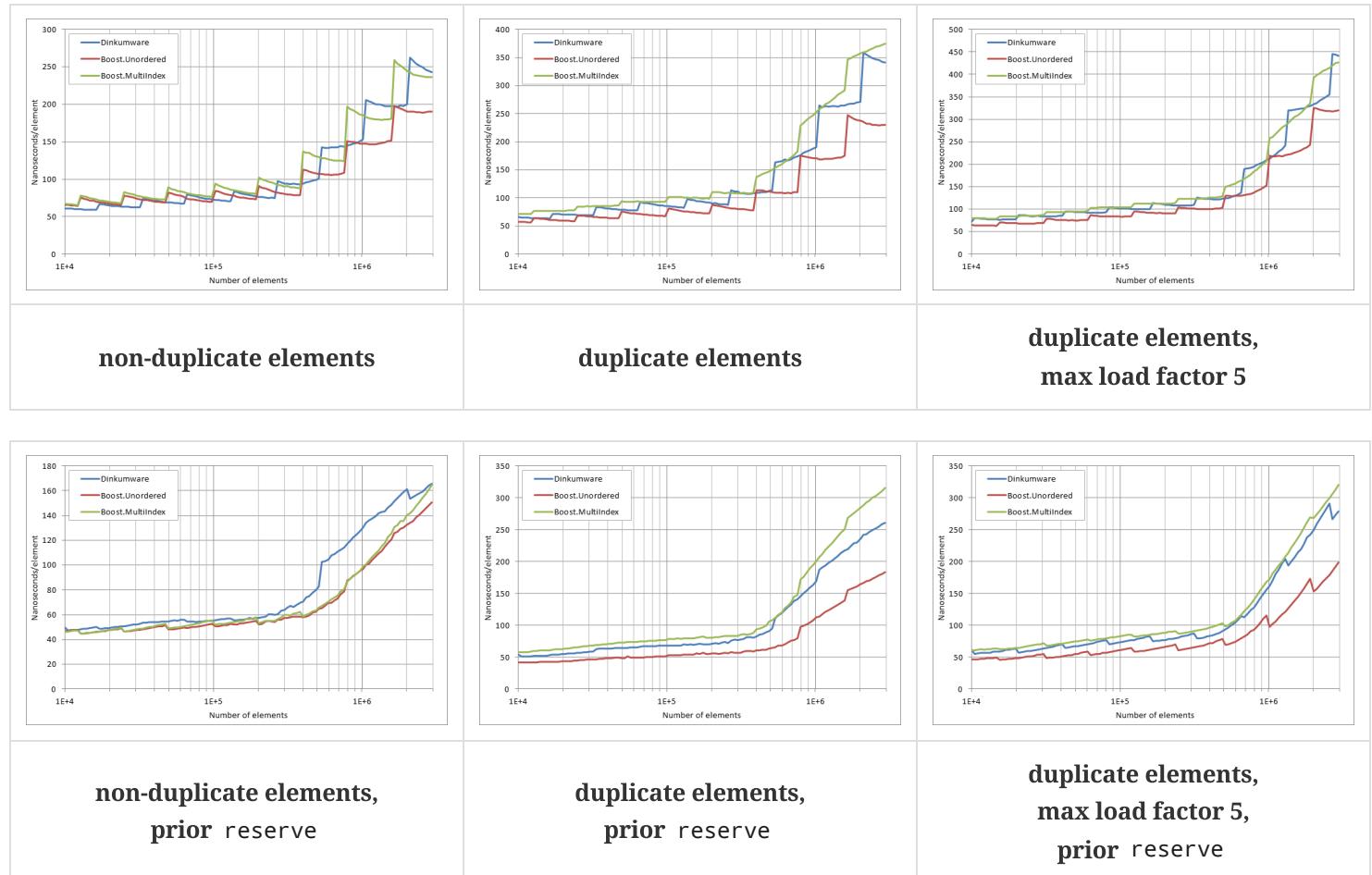
### Successful lookup



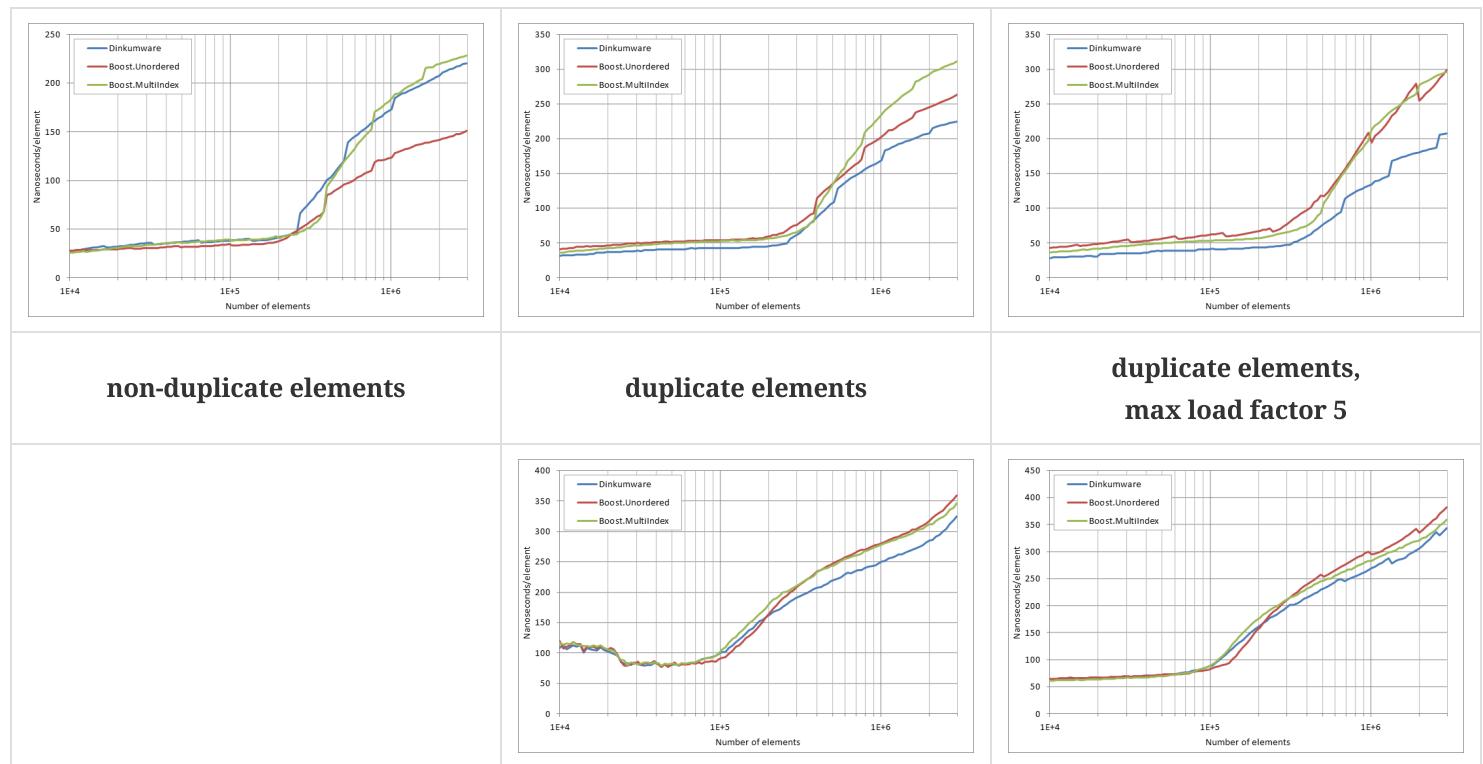
### Unsuccessful lookup



## Insertion



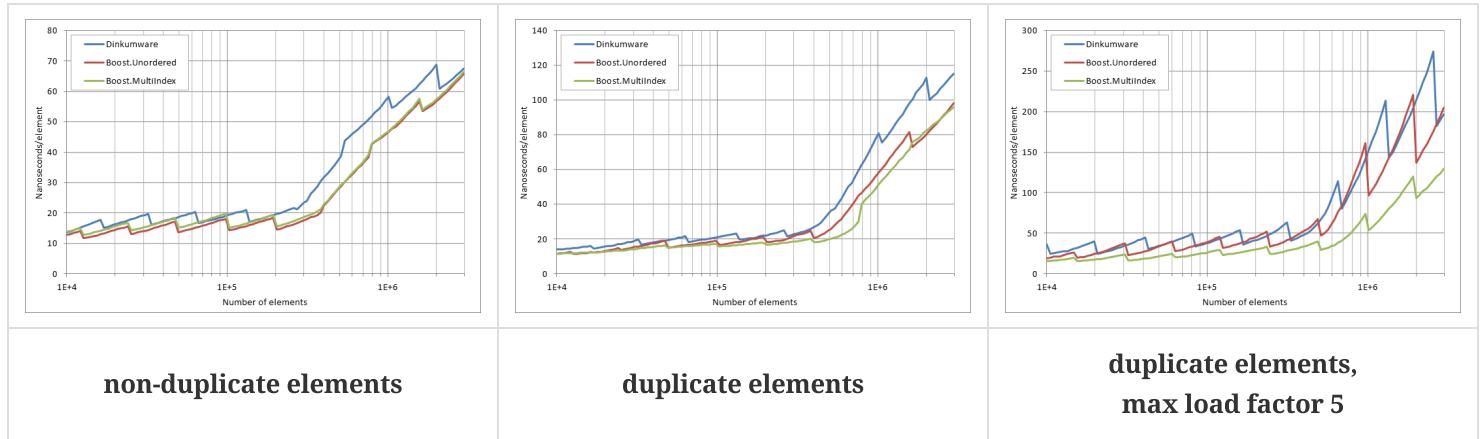
## Erasure



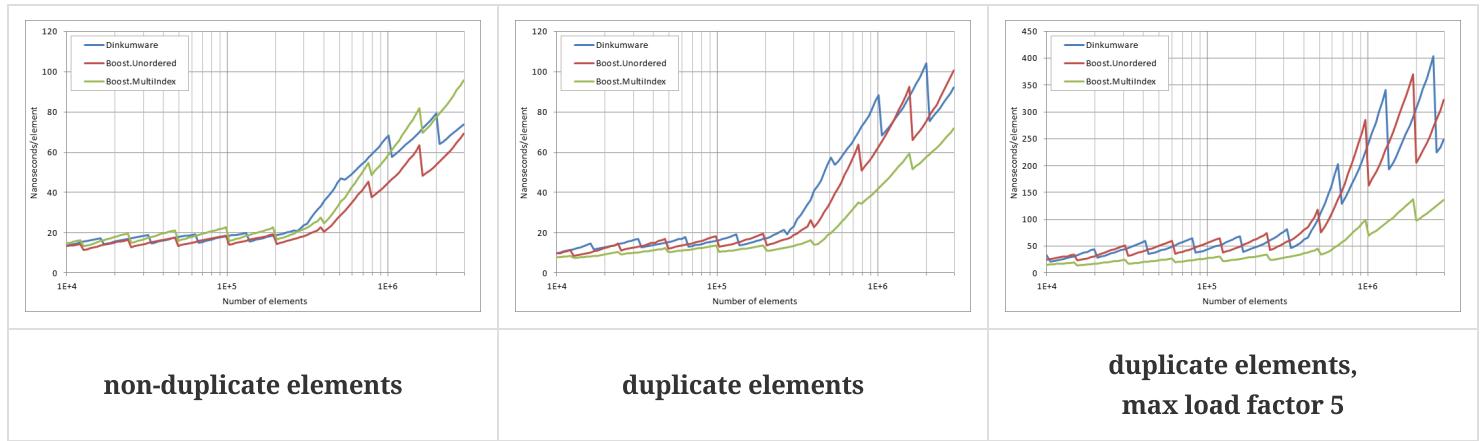
## by key, duplicate elements

## by key, duplicate elements, max load factor 5

### Successful lookup



### Unsuccessful lookup



### boost::unordered\_(flat | node)\_map

All benchmarks were created using:

- [`absl::flat\_hash\_map`](https://abseil.io/docs/cpp/guides/container) ([<uint64\\_t, uint64\\_t>](https://abseil.io/docs/cpp/guides/container))
- `boost::unordered_map<uint64_t, uint64_t>`
- `boost::unordered_flat_map<uint64_t, uint64_t>`
- `boost::unordered_node_map<uint64_t, uint64_t>`

The source code can be [found here](https://github.com/boostorg/boost_unordered_benchmarks/tree/boost_unordered_flat_map) ([https://github.com/boostorg/boost\\_unordered\\_benchmarks/tree/boost\\_unordered\\_flat\\_map](https://github.com/boostorg/boost_unordered_benchmarks/tree/boost_unordered_flat_map)).

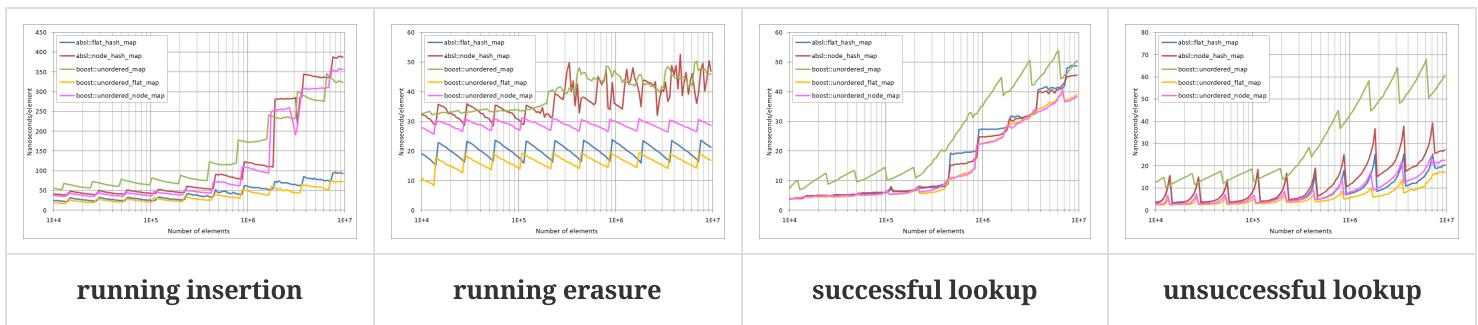
The insertion benchmarks insert `n` random values, where `n` is between 10,000 and 10 million.

The erasure benchmarks erase traverse the `n` elements and erase those with odd key (50% on average).

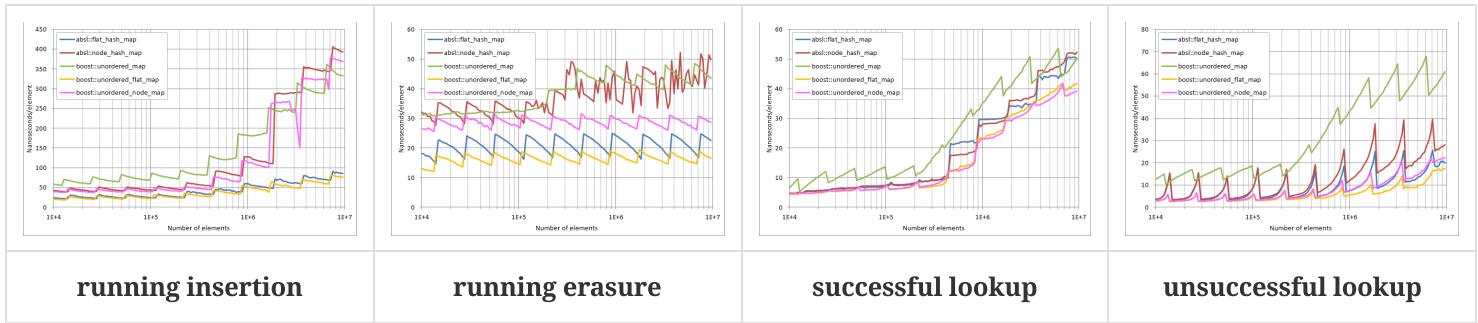
The successful lookup benchmarks are done by looking up all `n` values, in their original insertion order.

The unsuccessful lookup benchmarks use `n` randomly generated integers but using a different seed value.

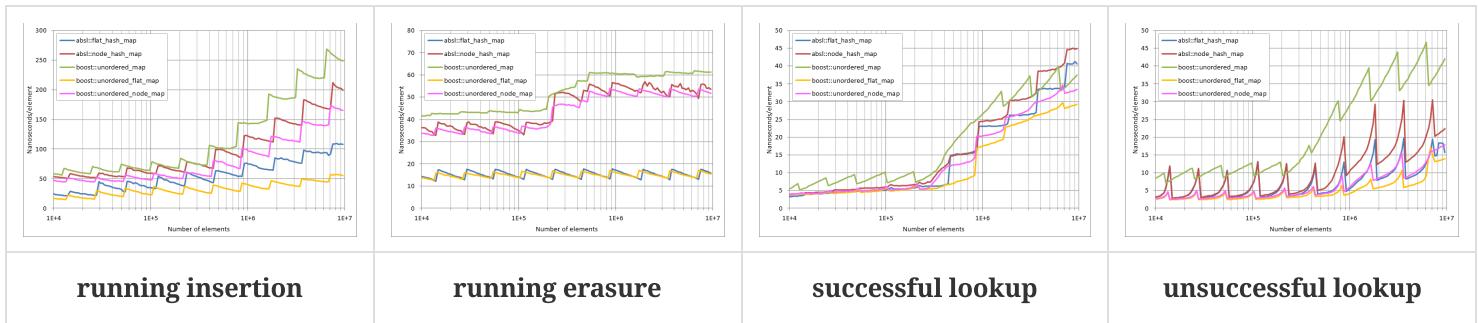
## GCC 12, x64



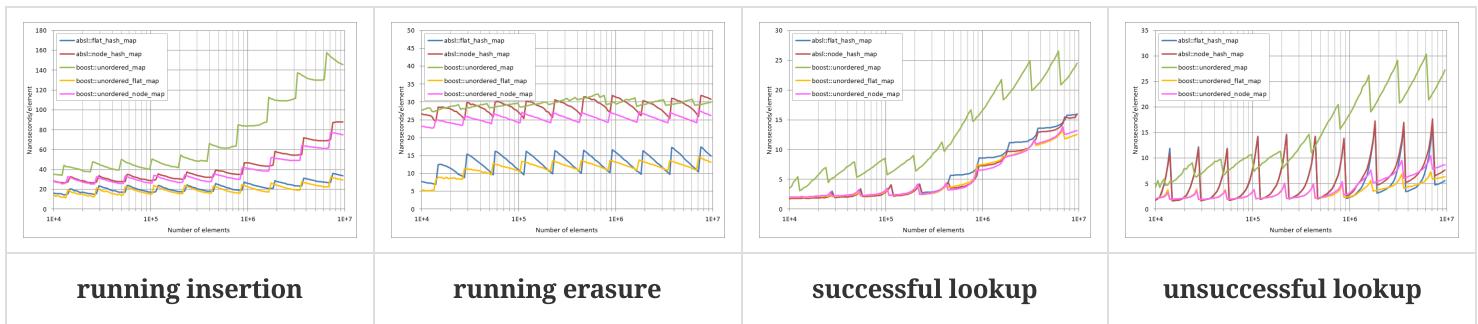
## Clang 15, x64



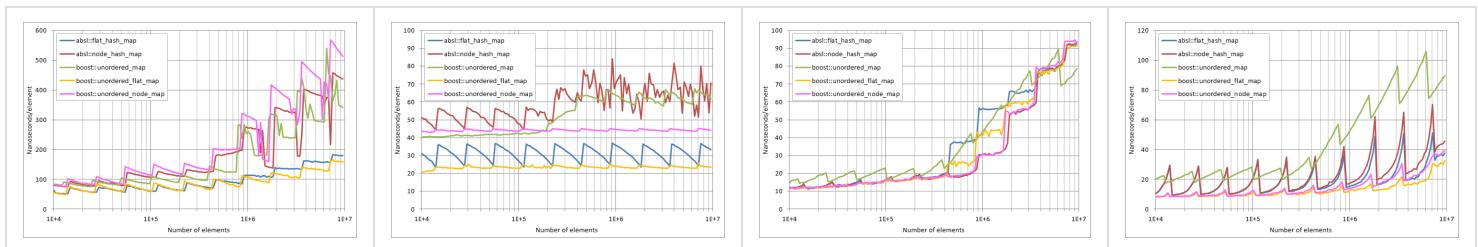
## Visual Studio 2022, x64



## Clang 12, ARM64

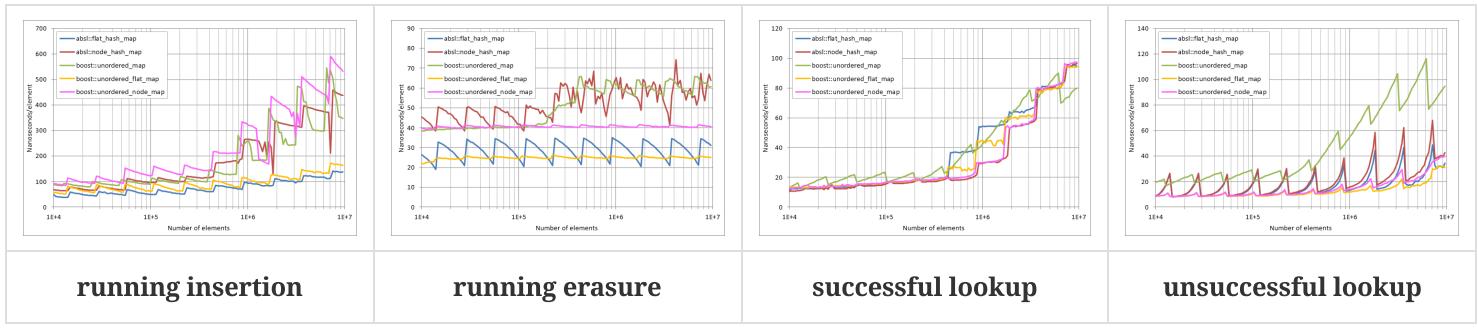


## GCC 12, x86

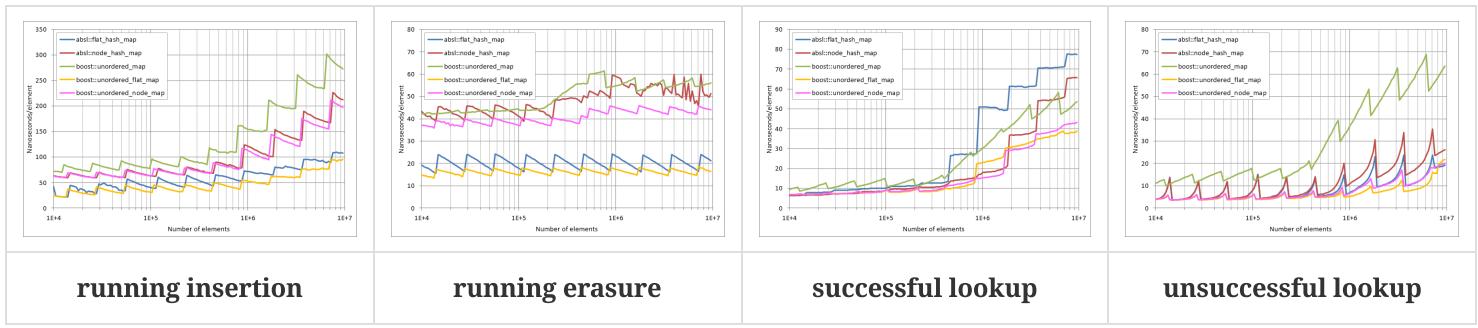


running insertion	running erasure	successful lookup	unsuccessful lookup
-------------------	-----------------	-------------------	---------------------

Clang 15, x86



Visual Studio 2022, x86



## boost::concurrent\_flat\_map

All benchmarks were created using:

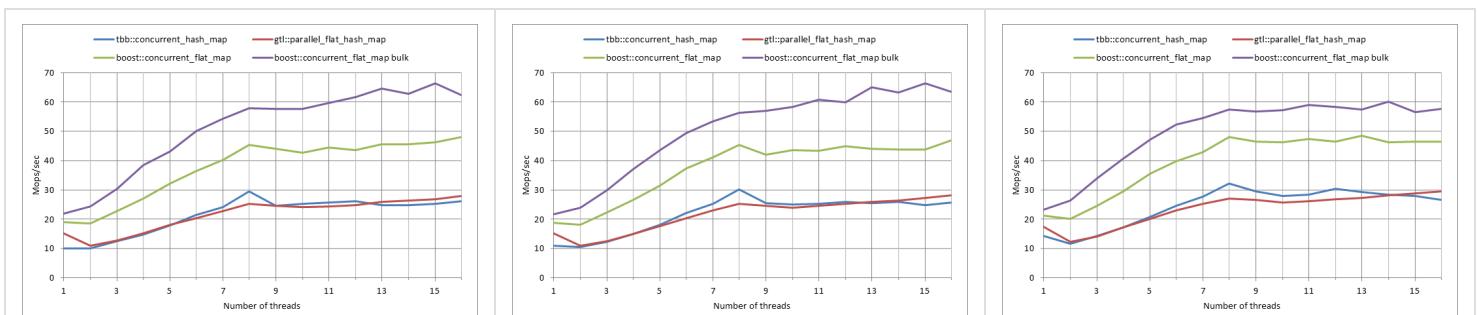
- [oneapi::tbb::concurrent hash map](#) ([https://spec.oneapi.io/versions/latest/elements/oneTBB/source/containers/concurrent\\_hash\\_map\\_cls.html](https://spec.oneapi.io/versions/latest/elements/oneTBB/source/containers/concurrent_hash_map_cls.html))<int , int>
- [gtl::parallel flat hash map](#) (<https://github.com/greg7mdp/gtl/blob/main/docs/phmap.md>)<int , int> with 64 submaps
- boost::concurrent\_flat\_map<int , int>

The source code can be [found here](#) ([https://github.com/boostorg/boost\\_unordered\\_benchmarks/tree/boost\\_concurrent\\_flat\\_map](https://github.com/boostorg/boost_unordered_benchmarks/tree/boost_concurrent_flat_map)).

The benchmarks exercise a number of threads  $T$  (between 1 and 16) concurrently performing operations randomly chosen among **update**, **successful lookup** and **unsuccessful lookup**. The keys used in the operations follow a [Zipf distribution](#) ([https://en.wikipedia.org/wiki/Zipf%27s\\_law#Formal\\_definition](https://en.wikipedia.org/wiki/Zipf%27s_law#Formal_definition)) with different *skew* parameters: the higher the skew, the more concentrated are the keys in the lower values of the covered range.

boost::concurrent\_flat\_map is exercised using both regular and bulk visitation: in the latter case, lookup keys are buffered in a local array and then processed at once each time the buffer reaches `bulk_visit_size`.

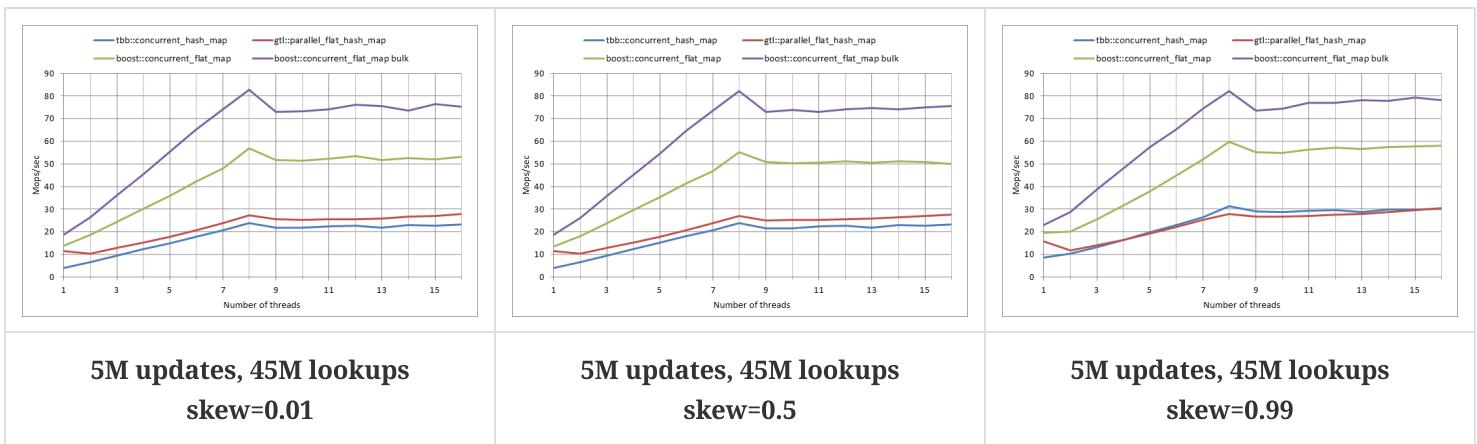
GCC 12, x64



500k updates, 4.5M lookups  
skew=0.01

500k updates, 4.5M lookups  
skew=0.5

500k updates, 4.5M lookups  
skew=0.99

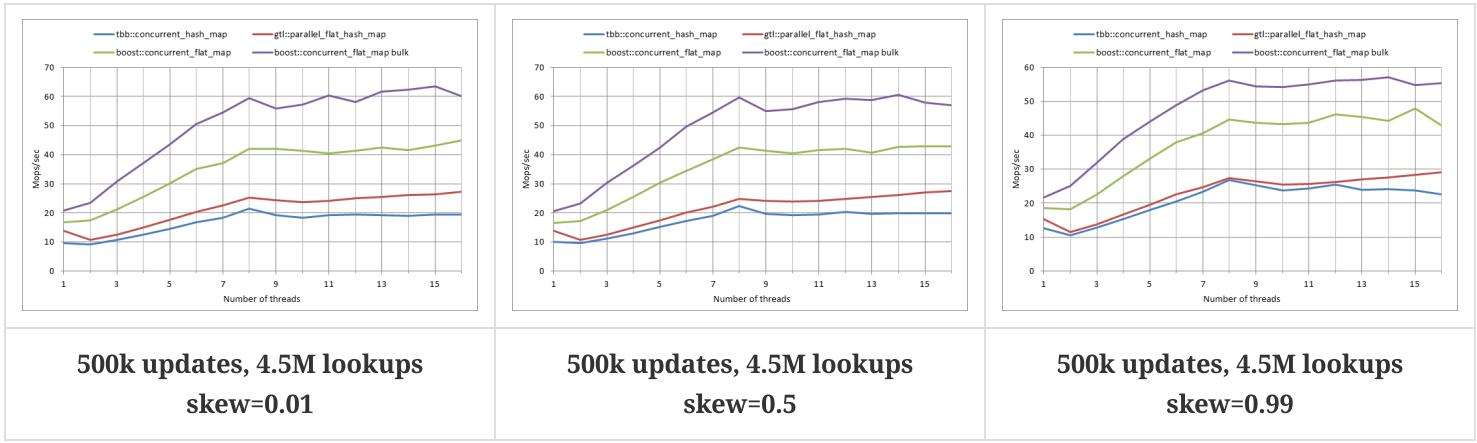


5M updates, 45M lookups  
skew=0.01

5M updates, 45M lookups  
skew=0.5

5M updates, 45M lookups  
skew=0.99

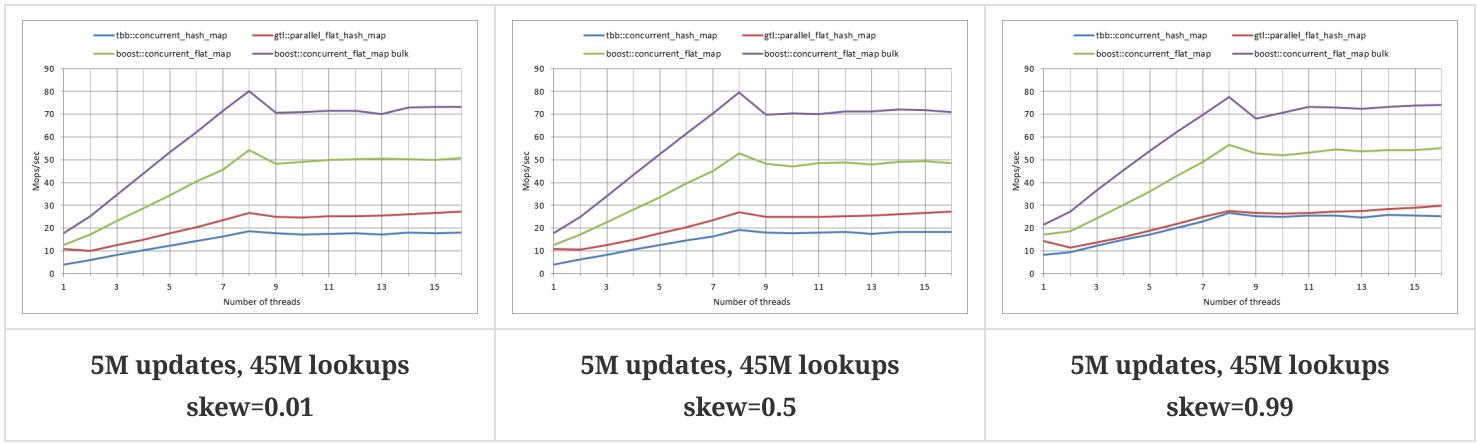
Clang 15, x64



500k updates, 4.5M lookups  
skew=0.01

500k updates, 4.5M lookups  
skew=0.5

500k updates, 4.5M lookups  
skew=0.99

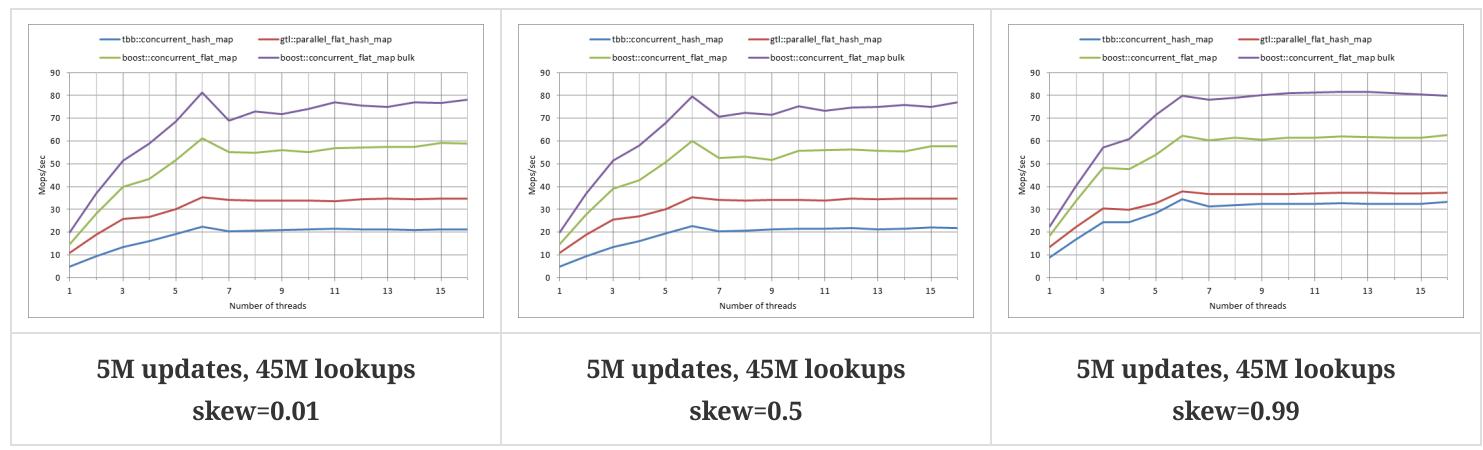
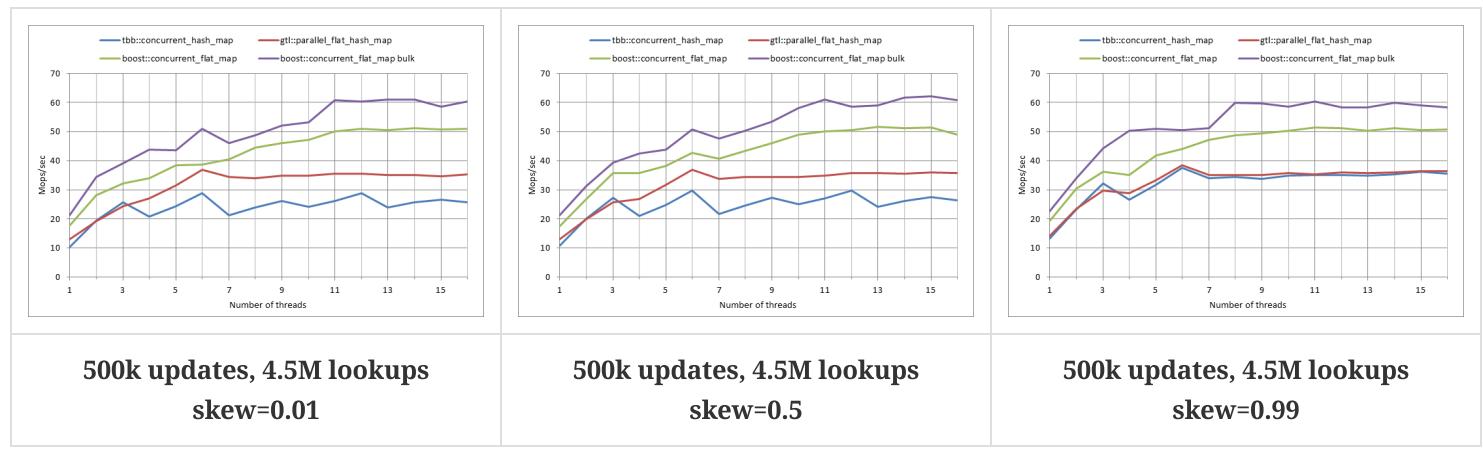


5M updates, 45M lookups  
skew=0.01

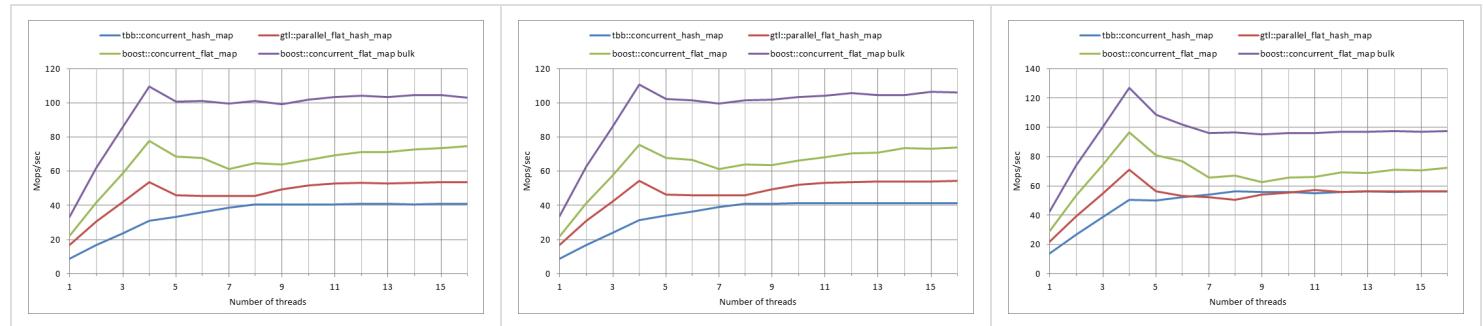
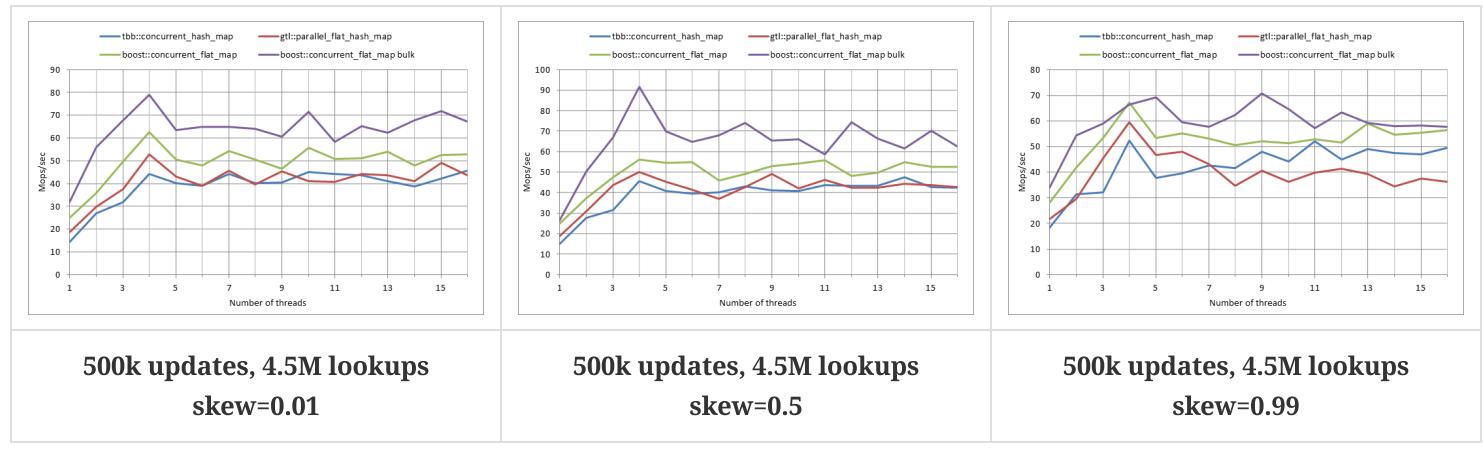
5M updates, 45M lookups  
skew=0.5

5M updates, 45M lookups  
skew=0.99

## Visual Studio 2022, x64



## Clang 12, ARM64

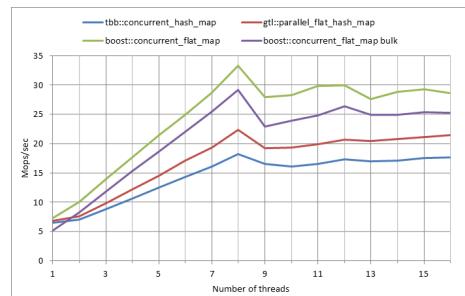
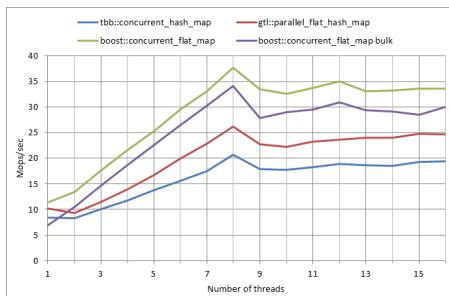
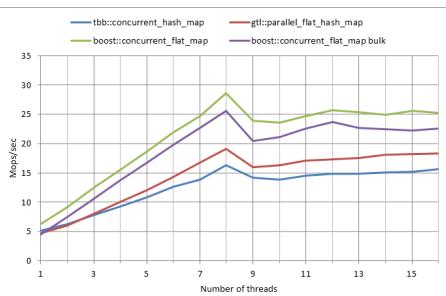


**5M updates, 45M lookups**  
skew=0.01

**5M updates, 45M lookups**  
skew=0.5

**5M updates, 45M lookups**  
skew=0.99

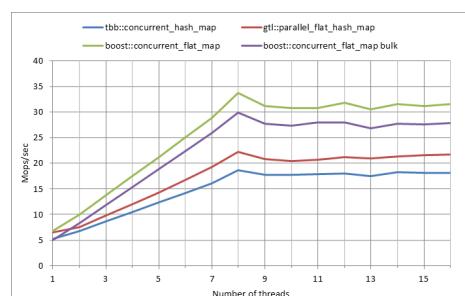
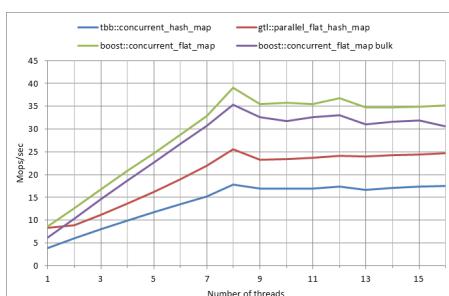
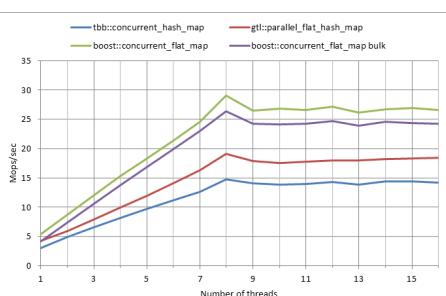
GCC 12, x86



**500k updates, 4.5M lookups**  
skew=0.01

**500k updates, 4.5M lookups**  
skew=0.5

**500k updates, 4.5M lookups**  
skew=0.99

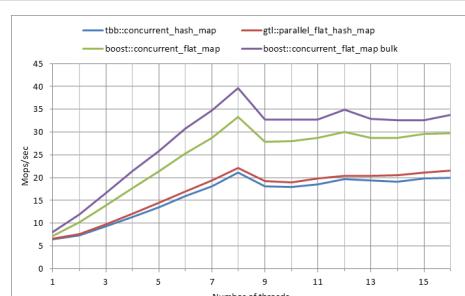
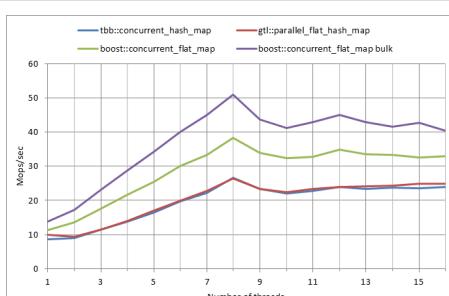
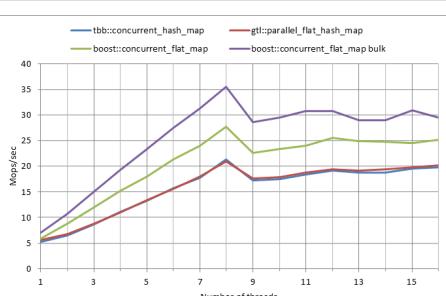


**5M updates, 45M lookups**  
skew=0.01

**5M updates, 45M lookups**  
skew=0.5

**5M updates, 45M lookups**  
skew=0.99

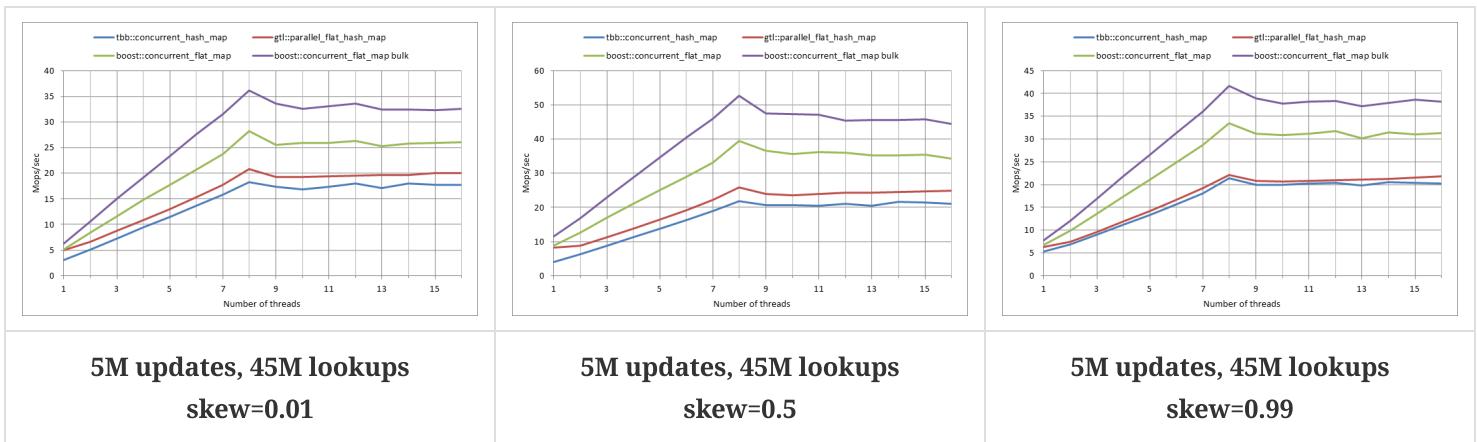
Clang 15, x86



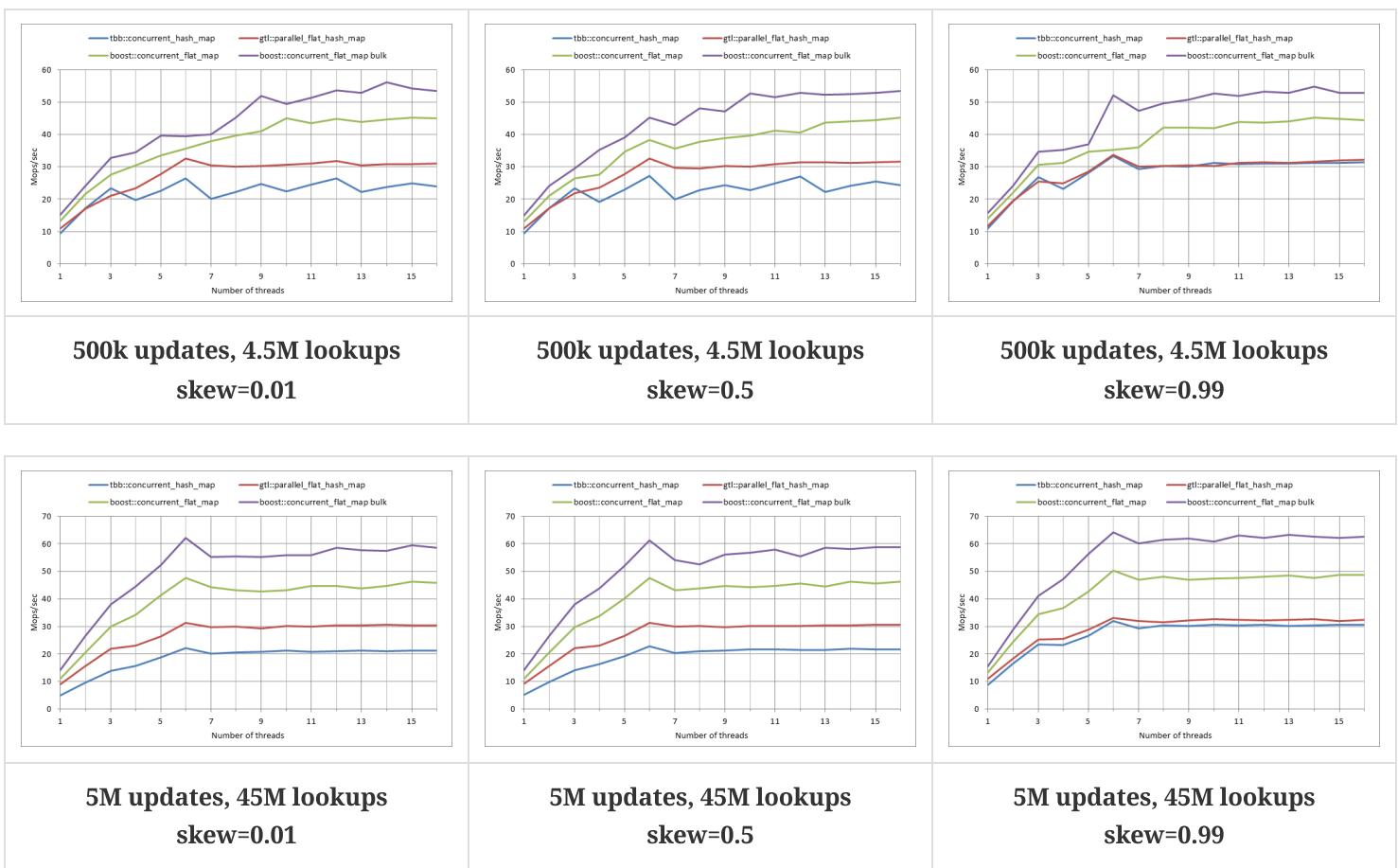
**500k updates, 4.5M lookups**  
skew=0.01

**500k updates, 4.5M lookups**  
skew=0.5

**500k updates, 4.5M lookups**  
skew=0.99



Visual Studio 2022, x86



## Implementation Rationale

### Closed-addressing Containers

`boost::unordered_[multi]set` and `boost::unordered_[multi]map` adhere to the standard requirements for unordered associative containers, so the interface was fixed. But there are still some implementation decisions to make. The priorities are conformance to the standard and portability.

The [Wikipedia article on hash tables](http://en.wikipedia.org/wiki/Hash_table) ([http://en.wikipedia.org/wiki/Hash\\_table](http://en.wikipedia.org/wiki/Hash_table)) has a good summary of the implementation issues for hash tables in general.

### Data Structure

By specifying an interface for accessing the buckets of the container the standard pretty much requires that the hash table uses closed addressing.

It would be conceivable to write a hash table that uses another method. For example, it could use open addressing, and use the lookup chain to act as a bucket but there are some serious problems with this:

- The standard requires that pointers to elements aren't invalidated, so the elements can't be stored in one array, but will need a layer of indirection instead - losing the efficiency and most of the memory gain, the main advantages of open addressing.
- Local iterators would be very inefficient and may not be able to meet the complexity requirements.
- There are also the restrictions on when iterators can be invalidated. Since open addressing degrades badly when there are a high number of collisions the restrictions could prevent a rehash when it's really needed. The maximum load factor could be set to a fairly low value to work around this - but the standard requires that it is initially set to 1.0.
- And since the standard is written with a eye towards closed addressing, users will be surprised if the performance doesn't reflect that.

So closed addressing is used.

### Number of Buckets

There are two popular methods for choosing the number of buckets in a hash table. One is to have a prime number of buckets, another is to use a power of 2.

Using a prime number of buckets, and choosing a bucket by using the modulus of the hash function's result will usually give a good result. The downside is that the required modulus operation is fairly expensive. This is what the containers used to do in most cases.

Using a power of 2 allows for much quicker selection of the bucket to use, but at the expense of losing the upper bits of the hash value. For some specially designed hash functions it is possible to do this and still get a good result but as the containers can take arbitrary hash functions this can't be relied on.

To avoid this a transformation could be applied to the hash function, for an example see [Thomas Wang's article on integer hash functions](http://web.archive.org/web/20121102023700/http://www.concentric.net/~Ttwang/tech/inthash.htm) (<http://web.archive.org/web/20121102023700/http://www.concentric.net/~Ttwang/tech/inthash.htm>). Unfortunately, a transformation like Wang's requires knowledge of the number of bits in the hash value, so it was only used when `size_t` was 64 bit.

Since release 1.79.0, [Fibonacci hashing](https://en.wikipedia.org/wiki/Hash_function#Fibonacci_hashing) ([https://en.wikipedia.org/wiki/Hash\\_function#Fibonacci\\_hashing](https://en.wikipedia.org/wiki/Hash_function#Fibonacci_hashing)) is used instead. With this implementation, the bucket number is determined by using  $(h * m) \gg (w - k)$ , where  $h$  is the hash value,  $m$  is  $2^w$  divided by the golden ratio,  $w$  is the word size (32 or 64), and  $2^k$  is the number of buckets. This provides a good compromise between speed and distribution.

Since release 1.80.0, prime numbers are chosen for the number of buckets in tandem with sophisticated modulo arithmetic. This removes the need for "mixing" the result of the user's hash function as was used for release 1.79.0.

### Open-addressing Containers

The C++ standard specification of unordered associative containers impose severe limitations on permissible implementations, the most important being that closed addressing is implicitly assumed. Slightly relaxing this specification opens up the possibility of providing container variations taking full advantage of open-addressing techniques.

The design of `boost::unordered_flat_set` / `unordered_node_set` and `boost::unordered_flat_map` / `unordered_node_map` has been guided by Peter Dimov's [Development Plan for Boost.Unordered](https://pdimov.github.io/articles/unordered_dev_plan.html) ([https://pdimov.github.io/articles/unordered\\_dev\\_plan.html](https://pdimov.github.io/articles/unordered_dev_plan.html)). We discuss here the most relevant principles.

## Hash Function

Given its rich functionality and cross-platform interoperability, `boost::hash` remains the default hash function of open-addressing containers. As it happens, `boost::hash` for integral and other basic types does not possess the statistical properties required by open addressing; to cope with this, we implement a post-mixing stage:

```
a ← h mulx C,  
h ← high(a) xor low(a),
```

where **mulx** is an *extended multiplication* (128 bits in 64-bit architectures, 64 bits in 32-bit environments), and **high** and **low** are the upper and lower halves of an extended word, respectively. In 64-bit architectures,  $C$  is the integer part of  $2^{64}\varphi$  ([https://en.wikipedia.org/wiki/Golden\\_ratio](https://en.wikipedia.org/wiki/Golden_ratio)), whereas in 32 bits  $C = 0xE817FB2Du$  has been obtained from [Steele and Vigna \(2021\)](#) (<https://arxiv.org/abs/2001.05304>).

When using a hash function directly suitable for open addressing, post-mixing can be opted out by via a dedicated `hash_is_avalanching` trait. `boost::hash` specializations for string types are marked as avalanching.

## Platform Interoperability

The observable behavior of `boost::unordered_flat_set` / `unordered_node_set` and `boost::unordered_flat_map` / `unordered_node_map` is deterministically identical across different compilers as long as their `std::size_t`s are the same size and the user-provided hash function and equality predicate are also interoperable —this includes elements being ordered in exactly the same way for the same sequence of operations.

Although the implementation internally uses SIMD technologies, such as [SSE2](#) (<https://en.wikipedia.org/wiki/SSE2>) and [Neon](#) ([https://en.wikipedia.org/wiki/ARM\\_architecture\\_family#Advanced SIMD\\_\(NEON\)](https://en.wikipedia.org/wiki/ARM_architecture_family#Advanced SIMD_(NEON))), when available, this does not affect interoperability. For instance, the behavior is the same for Visual Studio on an x64-mode Intel CPU with SSE2 and for GCC on an IBM s390x without any supported SIMD technology.

## Concurrent Containers

The same data structure used by Boost.Unordered open-addressing containers has been chosen also as the foundation of `boost::concurrent_flat_set` and `boost::concurrent_flat_map`:

- Open-addressing is faster than closed-addressing alternatives, both in non-concurrent and concurrent scenarios.
- Open-addressing layouts are eminently suitable for concurrent access and modification with minimal locking. In particular, the metadata array can be used for implementations of lookup that are lock-free up to the last step of actual element comparison.
- Layout compatibility with Boost.Unordered flat containers allows for fast transfer of all elements between `boost::concurrent_flat_map` and `boost::unordered_flat_map`, and vice versa.

## Hash Function and Platform Interoperability

Concurrent containers make the same decisions and provide the same guarantees as Boost.Unordered open-addressing containers with regards to hash function defaults and platform interoperability.

## Reference

### Class Template `unordered_map`

`boost::unordered_map` — An unordered associative container that associates unique keys with another value.

#### Synopsis

```

// #include <boost/unordered/unordered_map.hpp>

namespace boost {
    template<class Key,
              class T,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<std::pair<const Key, T>>>
    class unordered_map {
public:
    // types
    using key_type           = Key;
    using mapped_type         = T;
    using value_type          = std::pair<const Key, T>;
    using hasher              = Hash;
    using key_equal           = Pred;
    using allocator_type      = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer        = typename std::allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = std::size_t;
    using difference_type     = std::ptrdiff_t;

    using iterator             = implementation-defined;
    using const_iterator        = implementation-defined;
    using local_iterator        = implementation-defined;
    using const_local_iterator  = implementation-defined;
    using node_type             = implementation-defined;
    using insert_return_type   = implementation-defined;

    // construct/copy/destroy
    unordered_map();
    explicit unordered_map(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_map(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_map(const unordered_map& other);
    unordered_map(unordered_map&& other);
    template<class InputIterator>
    unordered_map(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_map(const Allocator& a);
    unordered_map(const unordered_map& other, const Allocator& a);
    unordered_map(unordered_map&& other, const Allocator& a);
    unordered_map(std::initializer_list<value_type> il,
                  size_type n = implementation-defined
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_map(size_type n, const allocator_type& a);
    unordered_map(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
    unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                  const allocator_type& a);
    unordered_map(std::initializer_list<value_type> il, const allocator_type& a);
}

```

```

unordered_map(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
unordered_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
            const allocator_type& a);
~unordered_map();
unordered_map& operator=(const unordered_map& other);
unordered_map& operator=(unordered_map&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_moveAssignable_v<Hash> &&
             boost::is_nothrow_moveAssignable_v<Pred>);
unordered_map& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
template<class P> std::pair<iterator, bool> insert(P&& obj);
iterator      insert(const_iterator hint, const value_type& obj);
iterator      insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type>);

template<class... Args>
    std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
    std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
    std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
    iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
template<class M>
    std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
    std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
    std::pair<iterator, bool> insert_or_assign(K&& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
template<class K, class M>
    iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);

node_type extract(const_iterator position);

```

```

node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator  erase(const_iterator first, const_iterator last);
void      quick_erase(const_iterator position);
void      erase_void(const_iterator position);
void      swap(unordered_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    iterator      find(CompatibleKey const& k, CompatibleHash const& hash,
                       CompatiblePredicate const& eq);
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    const_iterator find(CompatibleKey const& k, CompatibleHash const& hash,
                       CompatiblePredicate const& eq) const;
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>  equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator>  equal_range(const K& k) const;

// element access
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

```

```

template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
template<class K> size_type bucket(const K& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-key-type<InputIterator>>,
         class Pred = std::equal_to<iter-key-type<InputIterator>>,
         class Allocator = std::allocator<iter-to-alloc-type<InputIterator>>>
unordered_map(InputIterator, InputIterator, typename see below::size_type = see below,
             Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash, Pred,
                  Allocator>;

template<class Key, class T, class Hash = boost::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
unordered_map(std::initializer_list<std::pair<Key, T>>,
              typename see below::size_type = see below, Hash = Hash(),
              Pred = Pred(), Allocator = Allocator())
-> unordered_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_map(InputIterator, InputIterator, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_map(InputIterator, InputIterator, typename see below::size_type, Hash, Allocator)
-> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
             Allocator)
-> unordered_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

```

```

template<class Key, class T, class Allocator>
unordered_map(std::initializer_list<std::pair<Key, T>>, Allocator)
-> unordered_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Hash, class Allocator>
unordered_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type, Hash,
Allocator)
-> unordered_map<Key, T, Hash, std::equal_to<Key>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
const unordered_map<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
const unordered_map<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
unordered_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_map<K, T, H, P, A>::size_type
erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	Key must be <a href="#">Erasable</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container (i.e. <code>allocator_traits</code> can destroy it).
<i>T</i>	<i>T</i> must be <a href="#">Erasable</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container (i.e. <code>allocator_traits</code> can destroy it).
<i>Hash</i>	A unary function object type that acts a hash function for a <code>Key</code> . It takes a single argument of type <code>Key</code> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that implements an equivalence relation on values of type <code>Key</code> . A binary function object that induces an equivalence relation on values of type <code>Key</code> . It takes two arguments of type <code>Key</code> and returns a value of type <code>bool</code> .

## Allocator

An allocator whose value type is the same as the container's value type. Allocators using [fancy pointers](#) ([https://en.cppreference.com/w/cpp/named\\_req/Allocator#Fancy\\_pointers](https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers)) are supported.

The elements are organized into buckets. Keys with the same hash code are stored in the same bucket.

The number of buckets can be automatically increased by a call to insert, or as the result of calling rehash.

## Configuration macros

`BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0`

Globally define this macro to support loading of `unordered_map`s saved to a Boost.Serialization archive with a version of Boost prior to Boost 1.84.

## Typedefs

`typedef implementation-defined iterator;`

C++

An iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

`typedef implementation-defined const_iterator;`

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

`typedef implementation-defined local_iterator;`

C++

An iterator with the same value type, difference type and pointer and reference type as iterator.

A `local_iterator` object can be used to iterate through a single bucket.

`typedef implementation-defined const_local_iterator;`

C++

A constant iterator with the same value type, difference type and pointer and reference type as `const_iterator`.

A `const_local_iterator` object can be used to iterate through a single bucket.

`typedef implementation-defined node_type;`

C++

A class for holding extracted container elements, modelling [NodeHandle](#)

([https://en.cppreference.com/w/cpp/container/node\\_handle](https://en.cppreference.com/w/cpp/container/node_handle)).

---

**typedef** *implementation-defined* insert\_return\_type;

C++

A specialization of an internal class template:

```
template<class Iterator, class NodeType>
struct insert_return_type // name is exposition only
{
    Iterator position;
    bool inserted;
    NodeType node;
};
```

C++

with `Iterator = iterator` and `NodeType = node_type`.

---

## Constructors

### Default Constructor

`unordered_map()`

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate, `allocator_type()` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_map(size_type n,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor

C++

```
template<class InputIterator>
unordered_map(InputIterator f, InputIterator l,
             size_type n = implementation-defined,
             const hasher& hf = hasher(),
             const key_equal& eql = key_equal(),
             const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

C++

```
unordered_map(unordered_map const& other);
```

The copy constructor. Copies the contained elements, hash function, predicate, maximum load factor and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

---

## Move Constructor

C++

```
unordered_map(unordered_map&& other);
```

The move constructor.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move-constructible.

---

## Iterator Range Constructor with Allocator

C++

```
template<class InputIterator>
unordered_map(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Allocator Constructor

C++

```
explicit unordered_map(Allocator const& a);
```

Constructs an empty container, using allocator `a`.

## Copy Constructor with Allocator

C++

```
unordered_map(unordered_map const& other, Allocator const& a);
```

Constructs an container, copying `other`'s contained elements, hash function, predicate, maximum load factor, but using allocator `a`.

## Move Constructor with Allocator

C++

```
unordered_map(unordered_map&& other, Allocator const& a);
```

Construct a container moving `other`'s contained elements, and having the hash function, predicate and maximum load factor, but using allocate `a`.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move insertable.

## Initializer List Constructor

C++

```
unordered_map(std::initializer_list<value_type> il,
              size_type n = implementation-defined
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Bucket Count Constructor with Allocator

C++

```
unordered_map(size_type n, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate, `a` as the allocator and a maximum load factor of 1.0 .

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)  
([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

```
unordered_map(size_type n, hasher const& hf, allocator_type const& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#)  
([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)  
([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>
unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

```
unordered_map(std::initializer_list<value_type> il, const allocator_type& a);
```

C++

Constructs an empty container using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Allocator

```
unordered_map(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Hasher and Allocator

```
unordered_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
              const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Destructor

```
~unordered_map();
```

C++

**Note:** The destructor is applied to every element, and all memory is deallocated

---

### Assignment

#### Copy Assignment

```
unordered_map& operator=(unordered_map const& other);
```

C++

The assignment operator Copies the contained elements, hash function, predicate and maximum load factor but not the allocator.

If `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, the allocator is overwritten, if not the copied elements are created using the existing allocator.

**Requires:** `value_type` is copy constructible

---

## Move Assignment

```
unordered_map& operator=(unordered_map&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
        boost::is_nothrow_move_assignable_v<Hash> &&
        boost::is_nothrow_move_assignable_v<Pred>);
```

C++

The move assignment operator.

If `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`, the allocator is overwritten, if not the moved elements are created using the existing allocator.

**Requires:** `value_type` is move constructible.

---

## Initializer List Assignment

```
unordered_map& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All existing elements are either overwritten by the new elements or destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container and [CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable)).

## Iterators

### begin

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

### end

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

---

### cbegin

```
const_iterator cbegin() const noexcept;
```

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

## cend

```
const_iterator cend() const noexcept;
```

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

**Returns:** `size() == 0`

---

### size

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

**Returns:** `std::distance(begin(), end())`

---

### max\_size

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

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

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

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## emplace\_hint

```
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
```

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

`position` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Copy Insert

```
std::pair<iterator, bool> insert(const value_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Move Insert

```
std::pair<iterator, bool> insert(value_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Emplace Insert

```
template<class P> std::pair<iterator, bool> insert(P&& obj);
```

C++

Inserts an element into the container by performing `emplace(std::forward<P>(value))`.

Only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

## Copy Insert with Hint

C++

```
iterator insert(const_iterator hint, const value_type& obj);
```

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Move Insert with Hint

C++

```
iterator insert(const_iterator hint, value_type&& obj);
```

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Emplace Insert with Hint

```
template<class P> iterator insert(const_iterator hint, P&& obj);
```

Inserts an element into the container by performing `emplace_hint(hint, std::forward<P>(value))`.

Only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

`hint` is a suggestion to where the element should be inserted.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Insert Iterator Range

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Insert Initializer List

```
void insert(std::initializer_list<value_type>);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## try\_emplace

C++

```
template<class... Args>
std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
std::pair<iterator, bool> try_emplace(K&& k, Args&&... args)
```

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similar to emplace except the `value_type` is constructed using:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

C++

instead of emplace which simply forwards all arguments to `value_type`'s constructor.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## try\_emplace with Hint

```
template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
```

C++

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

`hint` is a suggestion to where the element should be inserted.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similar to `emplace_hint` except the `value_type` is constructed using:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

instead of `emplace_hint` which simply forwards all arguments to `value_type`'s constructor.

The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## insert\_or\_assign

```
template<class M>
std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
std::pair<iterator, bool> insert_or_assign(K&& k, M&& obj);
```

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## insert\_or\_assign with Hint

```
template<class M>
iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
template<class K, class M>
iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);
```

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

`hint` is a suggestion to where the element should be inserted.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Extract by Iterator

`node_type extract(const_iterator position);`

C++

Removes the element pointed to by `position`.

**Returns:** A `node_type` owning the element.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_multimap`.

---

## Extract by Key

`node_type extract(const key_type& k);`  
`template<class K> node_type extract(K&& k);`

C++

Removes an element with key equivalent to `k`.

**Returns:** A `node_type` owning the element if found, otherwise an empty `node_type`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_multimap`.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert with `node_handle`

```
insert_return_type insert(node_type&& nh);
```

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh` if and only if there is no element in the container with an equivalent key.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns an `insert_return_type` with: `inserted` equal to `false`, `position` equal to `end()` and `node` empty.

Otherwise if there was already an element with an equivalent key, returns an `insert_return_type` with: `inserted` equal to `false`, `position` pointing to a matching element and `node` contains the node from `nh`.

Otherwise if the insertion succeeded, returns an `insert_return_type` with: `inserted` equal to `true`, `position` pointing to the newly inserted element and `node` empty.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_multimap`.

---

## Insert with Hint and `node_handle`

```
iterator insert(const_iterator hint, node_type&& nh);
```

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh` if and only if there is no element in the container with an equivalent key.

If there is already an element in the container with an equivalent key has no effect on `nh` (i.e. `nh` still contains the node.)

`hint` is a suggestion to where the element should be inserted.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty returns `end()`.

If there was already an element in the container with an equivalent key returns an iterator pointing to that.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to hasher the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_multimap`.

## Erase by Position

```
iterator erase(iterator position);
iterator erase(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Returns:** The iterator following `position` before the erasure.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** In older versions this could be inefficient because it had to search through several buckets to find the position of the returned iterator. The data structure has been changed so that this is no longer the case, and the alternative erase methods have been deprecated.

## Erase by Key

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

C++

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Erase Range

```
iterator erase(const_iterator first, const_iterator last);
```

C++

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

---

## quick\_erase

```
void quick_erase(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## erase\_return\_void

```
void erase_return_void(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## swap

C++

```
void swap(unordered_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
```

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## clear

C++

```
void clear();
```

Erases all elements in the container.

**Postconditions:** `size() == 0`

**Throws:** Never throws an exception.

---

## merge

C++

```
template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
```

Attempt to "merge" two containers by iterating `source` and extracting any node in `source` that is not contained in `*this` and then inserting it into `*this`.

Because `source` can have a different hash function and key equality predicate, the key of each node in `source` is rehashed using `this->hash_function()` and then, if required, compared using `this->key_eq()`.

The behavior of this function is undefined if `this->get_allocator() != source.get_allocator()`.

This function does not copy or move any elements and instead simply relocates the nodes from `source` into `*this`.

- Notes:**
- Pointers and references to transferred elements remain valid.
  - Invalidates iterators to transferred elements.
  - Invalidates iterators belonging to `*this`.
  - Iterators to non-transferred elements in `source` remain valid.
- 

## Observers

### get\_allocator

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

C++

### hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

### key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
template<class K>
const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
iterator      find(CompatibleKey const& k, CompatibleHash const& hash,
                  CompatiblePredicate const& eq);
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
const_iterator find(CompatibleKey const& k, CompatibleHash const& hash,
                   CompatiblePredicate const& eq) const;
```

C++

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.

**Notes:** The templated overloads containing `CompatibleKey`, `CompatibleHash` and `CompatiblePredicate` are non-standard extensions which allow you to use a compatible hash function and equality predicate for a key of a different type in order to avoid an expensive type cast. In general, its use is not encouraged and instead the `K` member function templates should be used.

The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## count

```
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
```

C++

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

```
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>  equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator>  equal_range(const K& k) const;
```

C++

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## operator[]

C++

```
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);
```

**Effects:** If the container does not already contain an elements with a key equivalent to `k`, inserts the value `std::pair<key_type const, mapped_type>(k, mapped_type())`.

**Returns:** A reference to `x.second` where `x` is the element already in the container, or the newly inserted element with a key equivalent to `k`.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## at

C++

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;
```

**Returns:** A reference to `x.second` where `x` is the (unique) element whose key is equivalent to `k`.

**Throws:** An exception object of type `std::out_of_range` if no such element is present.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

C++

**Returns:** The number of buckets.

---

### max\_bucket\_count

```
size_type max_bucket_count() const noexcept;
```

C++

**Returns:** An upper bound on the number of buckets.

---

### bucket\_size

```
size_type bucket_size(size_type n) const;
```

C++

**Requires:**  $n < \text{bucket\_count}()$

**Returns:** The number of elements in bucket  $n$ .

---

### bucket

```
size_type bucket(const key_type& k) const;  
template<class K> size_type bucket(const K& k) const;
```

C++

**Returns:** The index of the bucket which would contain an element with key  $k$ .

**Postconditions:** The return value is less than `bucket_count()`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

### begin

```
local_iterator begin(size_type n);  
const_local_iterator begin(size_type n) const;
```

C++

**Requires:**  $n$  shall be in the range  $[0, \text{bucket\_count}())$ .

**Returns:** A local iterator pointing the first element in the bucket with index `n`.

---

end

C++

```
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

cbegin

C++

```
const_local_iterator cbegin(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A constant local iterator pointing the first element in the bucket with index `n`.

---

cend

C++

```
const_local_iterator cend(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A constant local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

Hash Policy

load\_factor

C++

```
float load_factor() const noexcept;
```

**Returns:** The average number of elements per bucket.

---

max\_load\_factor

C++

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the current maximum load factor.

## Set max\_load\_factor

C++

```
void max_load_factor(float z);
```

**Effects:** Changes the container's maximum load factor, using `z` as a hint.

---

## rehash

C++

```
void rehash(size_type n);
```

Changes the number of buckets so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## reserve

C++

```
void reserve(size_type n);
```

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`, or `a.rehash(1)` if `n > 0` and `a.max_load_factor() == std::numeric_limits<float>::infinity()`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

### *iter-key-type*

```
template<class InputIterator>
using iter-key-type = std::remove_const_t<
    std::tuple_element_t<0, iter-value-type<InputIterator>>>; // exposition only
```

### *iter-mapped-type*

```
template<class InputIterator>
using iter-mapped-type =
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

### *iter-to-alloc-type*

```
template<class InputIterator>
using iter-to-alloc-type = std::pair<
    std::add_const_t<std::tuple_element_t<0, iter-value-type<InputIterator>>>,
    std::tuple_element_t<1, iter-value-type<InputIterator>>>; // exposition only
```

## Equality Comparisons

### `operator==`

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

### `operator!=`

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y))));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_map<K, T, H, P, A>::size_type
erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

C++

## Serialization

`unordered_map`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_map` to an archive

Saves all the elements of an `unordered_map` `x` to an archive (XML archive) `ar`.

**Requires:** `std::remove_const<key_type>::type` and `std::remove_const<mapped_type>::type` are serializable (XML serializable), and they do support Boost.Serialization `save_construct_data / load_construct_data` protocol (automatically supported by [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

## Loading an `unordered_map` from an archive

Deletes all preexisting elements of an `unordered_map` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_map` `other` saved to the storage read by `ar`.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `(std::remove_const<key_type>::type&&, std::remove_const<mapped_type>::type&&)`. `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Note:** If the archive was saved using a release of Boost prior to Boost 1.84, the configuration macro `BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0` has to be globally defined for this operation to succeed; otherwise, an exception is thrown.

---

## Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be and `end()` iterator.

**Requires:** The `unordered_map` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

## Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_map` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template `unordered_multimap`

`boost::unordered_multimap` — An unordered associative container that associates keys with another value. The same key can be stored multiple times.

### Synopsis

```

// #include <boost/unordered/unordered_map.hpp>

namespace boost {
    template<class Key,
              class T,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<std::pair<const Key, T>>>
    class unordered_multimap {
public:
    // types
    using key_type           = Key;
    using mapped_type         = T;
    using value_type          = std::pair<const Key, T>;
    using hasher              = Hash;
    using key_equal           = Pred;
    using allocator_type      = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer        = typename std::allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = std::size_t;
    using difference_type     = std::ptrdiff_t;

    using iterator             = implementation-defined;
    using const_iterator        = implementation-defined;
    using local_iterator        = implementation-defined;
    using const_local_iterator   = implementation-defined;
    using node_type             = implementation-defined;

    // construct/copy/destroy
    unordered_multimap();
    explicit unordered_multimap(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_multimap(const unordered_multimap& other);
    unordered_multimap(unordered_multimap&& other);
    template<class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_multimap(const Allocator& a);
    unordered_multimap(const unordered_multimap& other, const Allocator& a);
    unordered_multimap(unordered_multimap&& other, const Allocator& a);
    unordered_multimap(std::initializer_list<value_type> il,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_multimap(size_type n, const allocator_type& a);
    unordered_multimap(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
    unordered_multimap(std::initializer_list<value_type> il, const allocator_type& a);
    unordered_multimap(std::initializer_list<value_type> il, size_type n,

```

```

        const allocator_type& a);
unordered_multimap(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                  const allocator_type& a);
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap& other);
unordered_multimap& operator=(unordered_multimap&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_moveAssignable_v<Hash> &&
             boost::is_nothrow_moveAssignable_v<Pred>);
unordered_multimap& operator=(std::initializer_list<value_type> il);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
template<class P> iterator insert(P&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type> il);

node_type extract(const_iterator position);
node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator   erase(iterator position);
iterator   erase(const_iterator position);
size_type  erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator   erase(const_iterator first, const_iterator last);
void      quick_erase(const_iterator position);
void      erase_void(const_iterator position);
void      swap(unordered_multimap& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>

```

```

void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
template<class K>
const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
iterator      find(CompatibleKey const& k, CompatibleHash const& hash,
                  CompatiblePredicate const& eq);
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
const_iterator find(CompatibleKey const& k, CompatibleHash const& hash,
                    CompatiblePredicate const& eq) const;
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
template<class K> size_type bucket(const K& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-key-type<InputIterator>>,
         class Pred = std::equal_to<iter-key-type<InputIterator>>,
         class Allocator = std::allocator<iter-to-alloc-type<InputIterator>>>
unordered_multimap(InputIterator, InputIterator, typename see below::size_type = see below,
                   Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                     Pred, Allocator>;

```

```

template<class Key, class T, class Hash = boost::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
unordered_multimap(std::initializer_list<std::pair<Key, T>>,
                  typename see below::size_type = see below, Hash = Hash(),
                  Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                      boost::hash<iter-key-type<InputIterator>>,
                      std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap(InputIterator, InputIterator, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                      boost::hash<iter-key-type<InputIterator>>,
                      std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Allocator)
-> unordered_multimap<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(std::initializer_list<std::pair<Key, T>>, Allocator)
-> unordered_multimap<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Hash, class Allocator>
unordered_multimap(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Hash, Allocator)
-> unordered_multimap<Key, T, Hash, std::equal_to<Key>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
          unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_multimap<K, T, H, P, A>::size_type
erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<p><code>Key</code> must be <a href="#">Erasable</a> (<a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a>) from the container (i.e. <code>allocator_traits</code> can destroy it).</p>
<i>T</i>	<p><code>T</code> must be <a href="#">Erasable</a> (<a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a>) from the container (i.e. <code>allocator_traits</code> can destroy it).</p>
<i>Hash</i>	<p>A unary function object type that acts a hash function for a <code>Key</code>. It takes a single argument of type <code>Key</code> and returns a value of type <code>std::size_t</code>.</p>
<i>Pred</i>	<p>A binary function object that implements an equivalence relation on values of type <code>Key</code>. A binary function object that induces an equivalence relation on values of type <code>Key</code>. It takes two arguments of type <code>Key</code> and returns a value of type <code>bool</code>.</p>
<i>Allocator</i>	<p>An allocator whose value type is the same as the container's value type. Allocators using <a href="#">fancy pointers</a> (<a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a>) are supported.</p>

The elements are organized into buckets. Keys with the same hash code are stored in the same bucket.

The number of buckets can be automatically increased by a call to `insert`, or as the result of calling `rehash`.

### Configuration macros

`BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0`

Globally define this macro to support loading of `unordered_multimap`s saved to a Boost.Serialization archive with a version of Boost prior to Boost 1.84.

### TypeDefs

C++

```
typedef implementation-defined iterator;
```

An iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

C++

```
typedef implementation-defined const_iterator;
```

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

---

```
typedef implementation-defined local_iterator;
```

C++

An iterator with the same value type, difference type and pointer and reference type as iterator.

A `local_iterator` object can be used to iterate through a single bucket.

---

```
typedef implementation-defined const_local_iterator;
```

C++

A constant iterator with the same value type, difference type and pointer and reference type as `const_iterator`.

A `const_local_iterator` object can be used to iterate through a single bucket.

---

```
typedef implementation-defined node_type;
```

C++

See `node_handle_map` for details.

---

## Constructors

### Default Constructor

```
unordered_multimap();
```

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate, `allocator_type()` as the allocator and a maximum load factor of 1.0.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_multimap(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor

```
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

```
unordered_multimap(const unordered_multimap& other);
```

C++

The copy constructor. Copies the contained elements, hash function, predicate, maximum load factor and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

---

## Move Constructor

```
unordered_multimap(unordered_multimap&& other);
```

C++

The move constructor.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move-constructible.

---

## Iterator Range Constructor with Allocator

```
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, const allocator_type& a);
```

C++

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of 1.0 and inserts the elements from `[f, 1)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

`explicit unordered_multimap(const Allocator& a);`

C++

Constructs an empty container, using allocator `a`.

---

## Copy Constructor with Allocator

`unordered_multimap(const unordered_multimap& other, const Allocator& a);`

C++

Constructs an container, copying `other`'s contained elements, hash function, predicate, maximum load factor, but using allocator `a`.

---

## Move Constructor with Allocator

`unordered_multimap(unordered_multimap&& other, const Allocator& a);`

C++

Construct a container moving `other`'s contained elements, and having the hash function, predicate and maximum load factor, but using allocate `a`.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move insertable.

---

## Initializer List Constructor

`unordered_multimap(std::initializer_list<value_type> il,  
size_type n = implementation-defined,  
const hasher& hf = hasher(),  
const key_equal& eql = key_equal(),  
const allocator_type& a = allocator_type());`

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Bucket Count Constructor with Allocator

C++

```
unordered_multimap(size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

C++

```
unordered_multimap(size_type n, const hasher& hf, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

C++

```
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

C++

```
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## initializer\_list Constructor with Allocator

C++

```
unordered_multimap(std::initializer_list<value_type> il, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## initializer\_list Constructor with Bucket Count and Allocator

C++

```
unordered_multimap(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

C++

```
unordered_multimap(std::initializer_list<value_type> il, size_type n, const hasher& hf,  
                  const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Destructor

C++

```
~unordered_multimap();
```

**Note:** The destructor is applied to every element, and all memory is deallocated

## Assignment

### Copy Assignment

C++

```
unordered_multimap& operator=(const unordered_multimap& other);
```

The assignment operator. Copies the contained elements, hash function, predicate and maximum load factor but not the allocator.

If `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, the allocator is overwritten, if not the copied elements are created using the existing allocator.

**Requires:** `value_type` is copy constructible

---

## Move Assignment

C++

```
unordered_multimap& operator=(unordered_multimap&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_move_assignable_v<Hash> &&
             boost::is_nothrow_move_assignable_v<Pred>);
```

The move assignment operator.

If `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`, the allocator is overwritten, if not the moved elements are created using the existing allocator.

**Requires:** `value_type` is move constructible.

---

## Initializer List Assignment

C++

```
unordered_multimap& operator=(std::initializer_list<value_type> il);
```

Assign from values in initializer list. All existing elements are either overwritten by the new elements or destroyed.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container and [CopyAssignable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable)).

---

## Iterators

### begin

C++

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

### end

```
iterator      end() noexcept;
const_iterator end() const noexcept;
```

**Returns:** An iterator which refers to the past-the-end value for the container.

---

## cbegin

```
const_iterator cbegin() const noexcept;
```

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

## cend

```
const_iterator cend() const noexcept;
```

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

**Returns:** `size() == 0`

---

### size

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

**Returns:** `std::distance(begin(), end())`

---

### max\_size

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

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

```
template<class... Args> iterator emplace(Args&&... args);
```

Inserts an object, constructed with the arguments `args`, in the container.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## emplace\_hint

```
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
```

Inserts an object, constructed with the arguments `args`, in the container.

`position` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Copy Insert

```
iterator insert(const value_type& obj);
```

Inserts `obj` in the container.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

- Returns:** An iterator pointing to the inserted element.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.
- Pointers and references to elements are never invalidated.
- 

## Move Insert

C++

```
iterator insert(value_type&& obj);
```

Inserts `obj` in the container.

- Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).
- Returns:** An iterator pointing to the inserted element.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.
- Pointers and references to elements are never invalidated.
- 

## Emplace Insert

C++

```
template<class P> iterator insert(P&& obj);
```

Inserts an element into the container by performing `emplace(std::forward<P>(value))`.

Only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

- Returns:** An iterator pointing to the inserted element.
- 

## Copy Insert with Hint

C++

```
iterator insert(const_iterator hint, const value_type& obj);
```

Inserts `obj` in the container.

`hint` is a suggestion to where the element should be inserted.

- Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

- Returns:** An iterator pointing to the inserted element.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.
- Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.
- Pointers and references to elements are never invalidated.
- 

## Move Insert with Hint

C++

```
iterator insert(const_iterator hint, value_type&& obj);
```

Inserts `obj` in the container.

`hint` is a suggestion to where the element should be inserted.

- Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).
- Returns:** An iterator pointing to the inserted element.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.
- Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.
- Pointers and references to elements are never invalidated.
- 

## Emplace Insert with Hint

C++

```
template<class P> iterator insert(const_iterator hint, P&& obj);
```

Inserts an element into the container by performing `emplace_hint(hint, std::forward<P>(value))`.

Only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

`hint` is a suggestion to where the element should be inserted.

- Returns:** An iterator pointing to the inserted element.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Insert Iterator Range

`template<class InputIterator> void insert(InputIterator first, InputIterator last);`

C++

Inserts a range of elements into the container.

**Requires:** `value_type` is [EmplaceConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Insert Initializer List

`void insert(std::initializer_list<value_type> il);`

C++

Inserts a range of elements into the container.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Extract by Iterator

`node_type extract(const_iterator position);`

C++

Removes the element pointed to by `position`.

**Returns:** A `node_type` owning the element.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_map`.

---

## Extract by Key

C++

```
node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
```

Removes an element with key equivalent to `k`.

**Returns:** A `node_type` owning the element if found, otherwise an empty `node_type`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_map`.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert with `node_handle`

C++

```
iterator insert(node_type&& nh);
```

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh`.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns `end()`.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_map`.

---

## Insert with Hint and `node_handle`

C++

```
iterator insert(const_iterator hint, node_type&& nh);
```

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh`.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns `end()`.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_map`.

---

## Erase by Position

C++

```
iterator erase(iterator position);
iterator erase(const_iterator position);
```

Erase the element pointed to by `position`.

**Returns:** The iterator following `position` before the erasure.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** In older versions this could be inefficient because it had to search through several buckets to find the position of the returned iterator. The data structure has been changed so that this is no longer the case, and the alternative erase methods have been deprecated.

---

## Erase by Key

C++

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The template<class `K`> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Erase Range

```
iterator erase(const_iterator first, const_iterator last);
```

C++

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

---

## quick\_erase

```
void quick_erase(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## erase\_return\_void

```
void erase_return_void(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## swap

```
void swap(unordered_multimap& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`

**Throws:** Never throws an exception.

---

## merge

```
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
```

C++

Attempt to "merge" two containers by iterating `source` and extracting all nodes in `source` and inserting them into `*this`.

Because `source` can have a different hash function and key equality predicate, the key of each node in `source` is rehashed using `this->hash_function()` and then, if required, compared using `this->key_eq()`.

The behavior of this function is undefined if `this->get_allocator() != source.get_allocator()`.

This function does not copy or move any elements and instead simply relocates the nodes from `source` into `*this`.

- Notes:**
- Pointers and references to transferred elements remain valid.
  - Invalidates iterators to transferred elements.
  - Invalidates iterators belonging to `*this`.
  - Iterators to non-transferred elements in `source` remain valid.

---

## Observers

### get\_allocator

`allocator_type get_allocator() const;`

C++

---

### hash\_function

`hasher hash_function() const;`

C++

- Returns:** The container's hash function.

---

### key\_eq

`key_equal key_eq() const;`

C++

- Returns:** The container's key equality predicate

---

## Lookup

### find

```

iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<typename ComparableKey, typename ComparableHash, typename ComparablePredicate>
    iterator      find(ComparableKey const& k, ComparableHash const& hash,
                        ComparablePredicate const& eq);
template<typename ComparableKey, typename ComparableHash, typename ComparablePredicate>
    const_iterator find(ComparableKey const& k, ComparableHash const& hash,
                        ComparablePredicate const& eq) const;

```

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.

**Notes:** The templated overloads containing `ComparableKey`, `ComparableHash` and `ComparablePredicate` are non-standard extensions which allow you to use a compatible hash function and equality predicate for a key of a different type in order to avoid an expensive type cast. In general, its use is not encouraged and instead the `K` member function templates should be used.

The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## count

```

size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;

```

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## contains

```

bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;

```

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

C++

```
std::pair<iterator, iterator>           equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>           equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;
```

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

C++

```
size_type bucket_count() const noexcept;
```

**Returns:** The number of buckets.

---

### max\_bucket\_count

C++

```
size_type max_bucket_count() const noexcept;
```

**Returns:** An upper bound on the number of buckets.

---

### bucket\_size

C++

```
size_type bucket_size(size_type n) const;
```

**Requires:** `n < bucket_count()`

**Returns:** The number of elements in bucket `n`.

---

## bucket

C++

```
size_type bucket(const key_type& k) const;  
template<class K> size_type bucket(const K& k) const;
```

**Returns:** The index of the bucket which would contain an element with key `k`.

**Postconditions:** The return value is less than `bucket_count()`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## begin

C++

```
local_iterator begin(size_type n);  
const_local_iterator begin(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A local iterator pointing the first element in the bucket with index `n`.

---

## end

C++

```
local_iterator end(size_type n);  
const_local_iterator end(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

## cbegin

C++

```
const_local_iterator cbegin(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count()]`.

**Returns:** A constant local iterator pointing the first element in the bucket with index `n`.

---

## cend

C++

```
const_local_iterator cend(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count())`.

**Returns:** A constant local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

## Hash Policy

### load\_factor

`float load_factor() const noexcept;`

C++

**Returns:** The average number of elements per bucket.

---

### max\_load\_factor

`float max_load_factor() const noexcept;`

C++

**Returns:** Returns the current maximum load factor.

---

### Set max\_load\_factor

`void max_load_factor(float z);`

C++

**Effects:** Changes the container's maximum load factor, using `z` as a hint.

---

### rehash

`void rehash(size_type n);`

C++

Changes the number of buckets so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

### reserve

`void reserve(size_type n);`

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`, or `a.rehash(1)` if `n > 0` and `a.max_load_factor() == std::numeric_limits<float>::infinity()`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

### *iter-key-type*

```
template<class InputIterator>
using iter-key-type = std::remove_const_t<
    std::tuple_element_t<0, iter-value-type<InputIterator>>>; // exposition only
```

### *iter-mapped-type*

```
template<class InputIterator>
using iter-mapped-type =
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

### *iter-to-alloc-type*

```
template<class InputIterator>
using iter-to-alloc-type = std::pair<
    std::add_const_t<std::tuple_element_t<0, iter-value-type<InputIterator>>>,
    std::tuple_element_t<1, iter-value-type<InputIterator>>>; // exposition only
```

## Equality Comparisons

## operator==

C++

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

Return `true` if `x.size() == y.size()` and for every equivalent key group in `x`, there is a group in `y` for the same key, which is a permutation (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

## operator!=

C++

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

Return `false` if `x.size() == y.size()` and for every equivalent key group in `x`, there is a group in `y` for the same key, which is a permutation (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

## Swap

C++

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
          unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));
```

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_multimap<K, T, H, P, A>::size_type
erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);
```

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
C++
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

## Serialization

`unordered_multimap`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_multimap` to an archive

Saves all the elements of an `unordered_multimap` `x` to an archive (XML archive) `ar`.

**Requires:** `std::remove_const<key_type>::type` and `std::remove_const<mapped_type>::type` are serializable (XML serializable), and they do support Boost.Serialization `save_construct_data / load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

### Loading an `unordered_multimap` from an archive

Deletes all preexisting elements of an `unordered_multimap` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_multimap` `other` saved to the storage read by `ar`.

**Requires:** `value_type` is [EmplaceConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `(std::remove_const<key_type>::type&&, std::remove_const<mapped_type>::type&&)`. `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Note:** If the archive was saved using a release of Boost prior to Boost 1.84, the configuration macro `BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0` has to be globally defined for this operation to succeed; otherwise, an exception is thrown.

### Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be an `end()` iterator.

**Requires:** The `unordered_multimap` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

### Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_multimap` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template `unordered_set`

`boost::unordered_set` — An unordered associative container that stores unique values.

### Synopsis

```

// #include <boost/unordered/unordered_set.hpp>

namespace boost {
    template<class Key,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<Key>>
    class unordered_set {
public:
    // types
    using key_type          = Key;
    using value_type         = Key;
    using hasher             = Hash;
    using key_equal          = Pred;
    using allocator_type     = Allocator;
    using pointer             = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer       = typename std::allocator_traits<Allocator>::const_pointer;
    using reference           = value_type&;
    using const_reference     = const value_type&;
    using size_type           = std::size_t;
    using difference_type     = std::ptrdiff_t;

    using iterator            = implementation-defined;
    using const_iterator       = implementation-defined;
    using local_iterator       = implementation-defined;
    using const_local_iterator = implementation-defined;
    using node_type            = implementation-defined;
    using insert_return_type   = implementation-defined;

    // construct/copy/destroy
    unordered_set();
    explicit unordered_set(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_set(const unordered_set& other);
    unordered_set(unordered_set&& other);
    template<class InputIterator>
        unordered_set(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_set(const Allocator& a);
    unordered_set(const unordered_set& other, const Allocator& a);
    unordered_set(unordered_set&& other, const Allocator& a);
    unordered_set(std::initializer_list<value_type> il,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_set(size_type n, const allocator_type& a);
    unordered_set(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
        unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
        unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
    unordered_set(std::initializer_list<value_type> il, const allocator_type& a);
    unordered_set(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
    unordered_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,

```

```

        const allocator_type& a);
~unordered_set();
unordered_set& operator=(const unordered_set& other);
unordered_set& operator=(unordered_set&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_move_assignable_v<Hash> &&
             boost::is_nothrow_move_assignable_v<Pred>);
unordered_set& operator=(std::initializer_list<value_type> il);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
template<class K> std::pair<iterator, bool> insert(K&& k);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class K> iterator insert(const_iterator hint, K&& k);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std:: initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
insert_return_type insert(node_type&& nh);
iterator      insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator  erase(const_iterator first, const_iterator last);
void      quick_erase(const_iterator position);
void      erase_void(const_iterator position);
void      swap(unordered_set& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);

```

```

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    iterator      find(CompatibleKey const& k, CompatibleHash const& hash,
                        CompatiblePredicate const& eq);
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    const_iterator find(CompatibleKey const& k, CompatibleHash const& hash,
                        CompatiblePredicate const& eq) const;
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
template<class K> size_type bucket(const K& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-value-type<InputIterator>>,
         class Pred = std::equal_to<iter-value-type<InputIterator>>,
         class Allocator = std::allocator<iter-value-type<InputIterator>>>
unordered_set(InputIterator, InputIterator, typename see below::size_type = see below,
             Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_set<iter-value-type<InputIterator>, Hash, Pred, Allocator>;

template<class T, class Hash = boost::hash<T>, class Pred = std::equal_to<T>,
         class Allocator = std::allocator<T>>
unordered_set(std::initializer_list<T>, typename see below::size_type = see below,

```

```

        Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_set<iter-value-type<InputIterator>,
           boost::hash<iter-value-type<InputIterator>>,
           std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_set(InputIterator, InputIterator, Allocator)
-> unordered_set<iter-value-type<InputIterator>,
           boost::hash<iter-value-type<InputIterator>>,
           std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type, Hash, Allocator)
-> unordered_set<iter-value-type<InputIterator>, Hash,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class T, class Allocator>
unordered_set(std::initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Allocator>
unordered_set(std::initializer_list<T>, Allocator)
-> unordered_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
unordered_set(std::initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_set<T, Hash, std::equal_to<T>, Allocator>;

// Equality Comparisons
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& x,
                  const unordered_set<Key, Hash, Pred, Alloc>& y);

template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& x,
                  const unordered_set<Key, Hash, Pred, Alloc>& y);

// swap
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
          unordered_set<Key, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class H, class P, class A, class Predicate>
typename unordered_set<K, H, P, A>::size_type
erase_if(unordered_set<K, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<i>Key</i> must be <u>Erasable</u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container (i.e. <code>allocator_traits</code> can destroy it).
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that implements an equivalence relation on values of type <i>Key</i> . A binary function object that induces an equivalence relation on values of type <i>Key</i> . It takes two arguments of type <i>Key</i> and returns a value of type <code>bool</code> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <u>fancy pointers</u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The elements are organized into buckets. Keys with the same hash code are stored in the same bucket.

The number of buckets can be automatically increased by a call to `insert`, or as the result of calling `rehash`.

### Configuration macros

`BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0`

Globally define this macro to support loading of `unordered_set`s saved to a Boost.Serialization archive with a version of Boost prior to Boost 1.84.

### Typedefs

`typedef implementation-defined iterator;`

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

`typedef implementation-defined const_iterator;`

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

`typedef implementation-defined local_iterator;`

C++

An iterator with the same value type, difference type and pointer and reference type as iterator.

A `local_iterator` object can be used to iterate through a single bucket.

---

```
typedef implementation-defined const_local_iterator;
```

C++

A constant iterator with the same value type, difference type and pointer and reference type as `const_iterator`.

A `const_local_iterator` object can be used to iterate through a single bucket.

---

```
typedef implementation-defined node_type;
```

C++

A class for holding extracted container elements, modelling `NodeHandle` ([https://en.cppreference.com/w/cpp/container/node\\_handle](https://en.cppreference.com/w/cpp/container/node_handle)).

---

```
typedef implementation-defined insert_return_type;
```

C++

A specialization of an internal class template:

```
template<class Iterator, class NodeType>
struct insert_return_type // name is exposition only
{
    Iterator position;
    bool inserted;
    NodeType node;
};
```

C++

with `Iterator = iterator` and `NodeType = node_type`.

---

## Constructors

### Default Constructor

```
unordered_set();
```

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate, `allocator_type()` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_set(size_type n,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor

```
template<class InputIterator>
unordered_set(InputIterator f, InputIterator l,
    size_type n = implementation-defined,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

```
unordered_set(const unordered_set& other);
```

The copy constructor. Copies the contained elements, hash function, predicate, maximum load factor and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

---

## Move Constructor

```
unordered_set(unordered_set&& other);
```

The move constructor.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move-constructible.

---

## Iterator Range Constructor with Allocator

C++

```
template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

C++

```
explicit unordered_set(const Allocator& a);
```

Constructs an empty container, using allocator `a`.

---

## Copy Constructor with Allocator

C++

```
unordered_set(const unordered_set& other, const Allocator& a);
```

Constructs an container, copying `other`'s contained elements, hash function, predicate, maximum load factor, but using allocator `a`.

---

## Move Constructor with Allocator

C++

```
unordered_set(unordered_set&& other, const Allocator& a);
```

Construct a container moving `other`'s contained elements, and having the hash function, predicate and maximum load factor, but using allocate `a`.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move insertable.

---

## Initializer List Constructor

```
unordered_set(std::initializer_list<value_type> il,
             size_type n = implementation-defined,
             const hasher& hf = hasher(),
             const key_equal& eql = key_equal(),
             const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of `1.0` and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Bucket Count Constructor with Allocator

```
unordered_set(size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Bucket Count Constructor with Hasher and Allocator

```
unordered_set(size_type n, const hasher& hf, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** hasher , key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

C++

```
template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
             const allocator_type& a);
```

Constructs an empty container with at least n buckets, using hf as the hash function, a as the allocator, with the default key equality predicate and a maximum load factor of 1.0 and inserts the elements from [f, l) into it.

**Requires:** key\_equal needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

C++

```
unordered_set(std::initializer_list<value_type> il, const allocator_type& a);
```

Constructs an empty container using a as the allocator and a maximum load factor of 1.0 and inserts the elements from il into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Allocator

C++

```
unordered_set(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

Constructs an empty container with at least n buckets, using a as the allocator and a maximum load factor of 1.0 and inserts the elements from il into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

C++

```
unordered_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,
             const allocator_type& a);
```

Constructs an empty container with at least n buckets, using hf as the hash function, a as the allocator and a maximum load factor of 1.0 and inserts the elements from il into it.

**Requires:** key\_equal needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Destructor

C++

```
~unordered_set();
```

**Note:** The destructor is applied to every element, and all memory is deallocated

---

## Assignment

### Copy Assignment

C++

```
unordered_set& operator=(const unordered_set& other);
```

The assignment operator. Copies the contained elements, hash function, predicate and maximum load factor but not the allocator.

If `Alloc::propagate_on_container_copy_assignment` exists and  
`Alloc::propagate_on_container_copy_assignment::value` is `true`, the allocator is overwritten, if not the copied elements are created using the existing allocator.

**Requires:** `value_type` is copy constructible

---

### Move Assignment

C++

```
unordered_set& operator=(unordered_set&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_moveAssignable_v<Hash> &&
             boost::is_nothrow_moveAssignable_v<Pred>);
```

The move assignment operator.

If `Alloc::propagate_on_container_move_assignment` exists and  
`Alloc::propagate_on_container_move_assignment::value` is `true`, the allocator is overwritten, if not the moved elements are created using the existing allocator.

**Requires:** `value_type` is move constructible.

---

### Initializer List Assignment

C++

```
unordered_set& operator=(std::initializer_list<value_type> il);
```

Assign from values in initializer list. All existing elements are either overwritten by the new elements or destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container and [CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable)).

---

## Iterators

### begin

```
iterator      begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

### end

```
iterator      end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

---

### cbegin

```
const_iterator cbegin() const noexcept;
```

C++

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

### cend

```
const_iterator cend() const noexcept;
```

C++

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

C++

**Returns:** `size() == 0`

---

### size

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

C++

**Returns:** std::distance(begin(), end())

---

## max\_size

size\_type max\_size() const noexcept;

C++

**Returns:** size() of the largest possible container.

---

## Modifiers

### emplace

template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent value.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent value.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

### emplace\_hint

template<class... Args> iterator emplace\_hint(const\_iterator position, Args&&... args);

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent value.

`position` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Copy Insert

`std::pair<iterator, bool> insert(const value_type& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Move Insert

`std::pair<iterator, bool> insert(value_type&& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The bool component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Transparent Insert

```
template<class K> std::pair<iterator, bool> insert(K&& k);
```

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.

**Returns:** The `bool` component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Copy Insert with Hint

```
iterator insert(const_iterator hint, const value_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Move Insert with Hint

`iterator insert(const_iterator hint, value_type&& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Transparent Insert with Hint

`template<class K> iterator insert(const_iterator hint, K&& k);`

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert Iterator Range

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Insert Initializer List

```
void insert(std::initializer_list<value_type>);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Extract by Iterator

`node_type extract(const_iterator position);`

C++

Removes the element pointed to by `position`.

**Returns:** A `node_type` owning the element.

**Notes:** In C++17 a node extracted using this method can be inserted into a compatible `unordered_multiset`, but that is not supported yet.

---

## Extract by Value

`node_type extract(const key_type& k);`  
`template<class K> node_type extract(K&& k);`

C++

Removes an element with key equivalent to `k`.

**Returns:** A `node_type` owning the element if found, otherwise an empty `node_type`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** In C++17 a node extracted using this method can be inserted into a compatible `unordered_multiset`, but that is not supported yet.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert with `node_handle`

`insert_return_type insert(node_type&& nh);`

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh` if and only if there is no element in the container with an equivalent key.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns an `insert_return_type` with: `inserted` equal to `false`, `position` equal to `end()` and `node` empty.

Otherwise if there was already an element with an equivalent key, returns an `insert_return_type` with: `inserted` equal to `false`, `position` pointing to a matching element and `node` contains the node from `nh`.

Otherwise if the insertion succeeded, returns an `insert_return_type` with: `inserted` equal to `true`, `position` pointing to the newly inserted element and `node` empty.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

In C++17 this can be used to insert a node extracted from a compatible `unordered_multiset`, but that is not supported yet.

---

## Insert with Hint and `node_handle`

iterator `insert(const_iterator hint, node_type&& nh);`

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh` if and only if there is no element in the container with an equivalent key.

If there is already an element in the container with an equivalent key has no effect on `nh` (i.e. `nh` still contains the node.)

`hint` is a suggestion to where the element should be inserted.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty returns `end()`.

If there was already an element in the container with an equivalent key returns an iterator pointing to that.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_multiset`.

---

## Erase by Position

C++  
`iterator erase(iterator position);  
iterator erase(const_iterator position);`

Erase the element pointed to by `position`.

**Returns:** The iterator following `position` before the erasure.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** In older versions this could be inefficient because it had to search through several buckets to find the position of the returned iterator. The data structure has been changed so that this is no longer the case, and the alternative erase methods have been deprecated.

---

## Erase by Value

C++  
`size_type erase(const key_type& k);  
template<class K> size_type erase(K&& k);`

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Erase Range

C++  
`iterator erase(const_iterator first, const_iterator last);`

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

---

## quick\_erase

```
void quick_erase(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## erase\_return\_void

```
void erase_return_void(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## swap

```
void swap(unordered_set& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
            boost::is_nothrow_swappable_v<Hash> &&
            boost::is_nothrow_swappable_v<Pred>);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`

**Throws:** Never throws an exception.

---

## merge

```
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
```

C++

Attempt to "merge" two containers by iterating `source` and extracting any node in `source` that is not contained in `*this` and then inserting it into `*this`.

Because `source` can have a different hash function and key equality predicate, the key of each node in `source` is rehashed using `this->hash_function()` and then, if required, compared using `this->key_eq()`.

The behavior of this function is undefined if `this->get_allocator() != source.get_allocator()`.

This function does not copy or move any elements and instead simply relocates the nodes from `source` into `*this`.

**Notes:**

- Pointers and references to transferred elements remain valid.
- Invalidates iterators to transferred elements.
- Invalidates iterators belonging to `*this`.
- Iterators to non-transferred elements in `source` remain valid.

## Observers

### get\_allocator

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

C++

---

### hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

### key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    iterator      find(CompatibleKey const& k, CompatibleHash const& hash,
                      CompatiblePredicate const& eq);
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
    const_iterator find(CompatibleKey const& k, CompatibleHash const& hash,
                      CompatiblePredicate const& eq) const;
```

C++

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.

**Notes:** The templated overloads containing `CompatibleKey`, `CompatibleHash` and `CompatiblePredicate` are non-standard extensions which allow you to use a compatible hash function and equality predicate for a key of a different type in order to avoid an expensive type cast. In general, its use is not encouraged and instead the `K` member function templates should be used.

The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## count

```
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
```

C++

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

```
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>  equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator>  equal_range(const K& k) const;
```

C++

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

C++

**Returns:** The number of buckets.

---

## max\_bucket\_count

```
size_type max_bucket_count() const noexcept;
```

C++

**Returns:** An upper bound on the number of buckets.

---

## bucket\_size

```
size_type bucket_size(size_type n) const;
```

C++

**Requires:**  $n < \text{bucket\_count}()$

**Returns:** The number of elements in bucket  $n$ .

---

## bucket

```
size_type bucket(const key_type& k) const;  
template<class K> size_type bucket(const K& k) const;
```

C++

**Returns:** The index of the bucket which would contain an element with key  $k$ .

**Postconditions:** The return value is less than `bucket_count()`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## begin

```
local_iterator begin(size_type n);  
const_local_iterator begin(size_type n) const;
```

C++

**Requires:**  $n$  shall be in the range `[0, bucket_count())`.

**Returns:** A local iterator pointing the first element in the bucket with index  $n$ .

---

## end

```
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count())`.

**Returns:** A local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

## cbegin

```
const_local_iterator cbegin(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count())`.

**Returns:** A constant local iterator pointing the first element in the bucket with index `n`.

---

## cend

```
const_local_iterator cend(size_type n) const;
```

**Requires:** `n` shall be in the range `[0, bucket_count())`.

**Returns:** A constant local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

## Hash Policy

### load\_factor

```
float load_factor() const noexcept;
```

**Returns:** The average number of elements per bucket.

---

### max\_load\_factor

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the current maximum load factor.

---

### Set max\_load\_factor

```
void max_load_factor(float z);
```

**Effects:** Changes the container's maximum load factor, using `z` as a hint.

---

## rehash

C++

```
void rehash(size_type n);
```

Changes the number of buckets so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## reserve

C++

```
void reserve(size_type n);
```

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`, or `a.rehash(1)` if `n > 0` and `a.max_load_factor() == std::numeric_limits<float>::infinity()`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

## iter-value-type

```
template<class InputIterator>
using iter_value_type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

## Equality Comparisons

### operator==

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& x,
                  const unordered_set<Key, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

### operator!=

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& x,
                  const unordered_set<Key, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

## Swap

```
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
          unordered_set<Key, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

## erase\_if

C++

```
template<class K, class H, class P, class A, class Predicate>
typename unordered_set<K, H, P, A>::size_type
erase_if(unordered_set<K, H, P, A>& c, Predicate pred);
```

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
C++
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

## Serialization

`unordered_set`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_set` to an archive

Saves all the elements of an `unordered_set` `x` to an archive (XML archive) `ar`.

**Requires:** `value_type` is serializable (XML serializable), and it supports [Boost.Serialization](#) `save_construct_data` / `load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

### Loading an `unordered_set` from an archive

Deletes all preexisting elements of an `unordered_set` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_set` `other` saved to the storage read by `ar`.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)). `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Note:** If the archive was saved using a release of Boost prior to Boost 1.84, the configuration macro `BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0` has to be globally defined for this operation to succeed; otherwise, an exception is thrown.

---

## Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be and `end()` iterator.

**Requires:** The `unordered_set` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

## Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_set` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template `unordered_multiset`

`boost::unordered_multiset` — An unordered associative container that stores values. The same key can be stored multiple times.

### Synopsis

```

// #include <boost/unordered/unordered_set.hpp>

namespace boost {
    template<class Key,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<Key>>
    class unordered_multiset {
public:
    // types
    using key_type           = Key;
    using value_type          = Key;
    using hasher              = Hash;
    using key_equal           = Pred;
    using allocator_type      = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer        = typename std::allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = std::size_t;
    using difference_type     = std::ptrdiff_t;

    using iterator             = implementation-defined;
    using const_iterator        = implementation-defined;
    using local_iterator        = implementation-defined;
    using const_local_iterator  = implementation-defined;
    using node_type             = implementation-defined;

    // construct/copy/destroy
    unordered_multiset();
    explicit unordered_multiset(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_multiset(const unordered_multiset& other);
    unordered_multiset(unordered_multiset&& other);
    template<class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_multiset(const Allocator& a);
    unordered_multiset(const unordered_multiset& other, const Allocator& a);
    unordered_multiset(unordered_multiset&& other, const Allocator& a);
    unordered_multiset(std::initializer_list<value_type> il,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_multiset(size_type n, const allocator_type& a);
    unordered_multiset(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
    unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
    unordered_multiset(std::initializer_list<value_type> il, const allocator_type& a);
    unordered_multiset(std::initializer_list<value_type> il, size_type n,
                      const allocator_type& a)
    unordered_multiset(std::initializer_list<value_type> il, size_type n, const hasher& hf,

```

```

        const allocator_type& a);
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset& other);
unordered_multiset& operator=(unordered_multiset&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_move_assignable_v<Hash> &&
             boost::is_nothrow_move_assignable_v<Pred>);
unordered_multiset& operator=(std::initializer_list<value_type> il);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type> il);

node_type extract(const_iterator position);
node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
template<class K> size_type erase(K&& x);
iterator  erase(const_iterator first, const_iterator last);
void      quick_erase(const_iterator position);
void      erase_void(const_iterator position);
void      swap(unordered_multiset&)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
             boost::is_nothrow_swappable_v<Hash> &&
             boost::is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;

```

```

key_equal key_eq() const;

// set operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
template<class K>
const_iterator find(const K& k) const;
template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
iterator      find(CompatibleKey const&, CompatibleHash const&,
                  CompatiblePredicate const&);

template<typename CompatibleKey, typename CompatibleHash, typename CompatiblePredicate>
const_iterator find(CompatibleKey const&, CompatibleHash const&,
                    CompatiblePredicate const&) const;
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
template<class K> size_type bucket(const K& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-value-type<InputIterator>>,
         class Pred = std::equal_to<iter-value-type<InputIterator>>,
         class Allocator = std::allocator<iter-value-type<InputIterator>>>
unordered_multiset(InputIterator, InputIterator, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multiset<iter-value-type<InputIterator>, Hash, Pred, Allocator>;

template<class T, class Hash = boost::hash<T>, class Pred = std::equal_to<T>,
         class Allocator = std::allocator<T>>
unordered_multiset(std::initializer_list<T>, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multiset<T, Hash, Pred, Allocator>;

```

```

template<class InputIterator, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_multiset<iter-value-type<InputIterator>,
    boost::hash<iter-value-type<InputIterator>>,
    std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_multiset(InputIterator, InputIterator, Allocator)
-> unordered_multiset<iter-value-type<InputIterator>,
    boost::hash<iter-value-type<InputIterator>>,
    std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_multiset<iter-value-type<InputIterator>, Hash,
    std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class T, class Allocator>
unordered_multiset(std::initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_multiset<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Allocator>
unordered_multiset(std::initializer_list<T>, Allocator)
-> unordered_multiset<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
unordered_multiset(std::initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_multiset<T, Hash, std::equal_to<T>, Allocator>;

// Equality Comparisons
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& x,
                  const unordered_multiset<Key, Hash, Pred, Alloc>& y);

template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& x,
                  const unordered_multiset<Key, Hash, Pred, Alloc>& y);

// swap
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y))));

// Erasure
template<class K, class H, class P, class A, class Predicate>
typename unordered_multiset<K, H, P, A>::size_type
erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	Key must be <u>Erasable</u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container (i.e. allocator_traits can destroy it).
------------	--

<i>Hash</i>	A unary function object type that acts a hash function for a <code>Key</code> . It takes a single argument of type <code>Key</code> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that implements an equivalence relation on values of type <code>Key</code> . A binary function object that induces an equivalence relation on values of type <code>Key</code> . It takes two arguments of type <code>Key</code> and returns a value of type <code>bool</code> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <a href="#">fancy pointers</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The elements are organized into buckets. Keys with the same hash code are stored in the same bucket and elements with equivalent keys are stored next to each other.

The number of buckets can be automatically increased by a call to `insert`, or as the result of calling `rehash`.

### Configuration macros

`BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0`

Globally define this macro to support loading of `unordered_multiset`s saved to a Boost.Serialization archive with a version of Boost prior to Boost 1.84.

### Typedefs

`typedef implementation-defined iterator;`

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

`typedef implementation-defined const_iterator;`

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

`typedef implementation-defined local_iterator;`

C++

An iterator with the same value type, difference type and pointer and reference type as iterator.

A `local_iterator` object can be used to iterate through a single bucket.

---

```
typedef implementation-defined const_local_iterator;
```

C++

A constant iterator with the same value type, difference type and pointer and reference type as const\_iterator.

A const\_local\_iterator object can be used to iterate through a single bucket.

---

```
typedef implementation-defined node_type;
```

C++

See node\_handle\_set for details.

---

## Constructors

### Default Constructor

```
unordered_multiset();
```

C++

Constructs an empty container using hasher() as the hash function, key\_equal() as the key equality predicate, allocator\_type() as the allocator and a maximum load factor of 1.0 .

**Postconditions:** size() == 0

**Requires:** If the defaults are used, hasher , key\_equal and allocator\_type need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_multiset(size_type n,  
                           const hasher& hf = hasher(),  
                           const key_equal& eql = key_equal(),  
                           const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least n buckets, using hf as the hash function, eql as the key equality predicate, a as the allocator and a maximum load factor of 1.0 .

**Postconditions:** size() == 0

**Requires:** If the defaults are used, hasher , key\_equal and allocator\_type need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Iterator Range Constructor

```
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

```
unordered_multiset(const unordered_multiset& other);
```

The copy constructor. Copies the contained elements, hash function, predicate, maximum load factor and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

---

## Move Constructor

```
unordered_multiset(unordered_multiset&& other);
```

The move constructor.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move-constructible.

---

## Iterator Range Constructor with Allocator

```
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of 1.0 and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

C++

```
explicit unordered_multiset(const Allocator& a);
```

Constructs an empty container, using allocator `a`.

---

## Copy Constructor with Allocator

C++

```
unordered_multiset(const unordered_multiset& other, const Allocator& a);
```

Constructs an container, copying `other`'s contained elements, hash function, predicate, maximum load factor, but using allocator `a`.

---

## Move Constructor with Allocator

C++

```
unordered_multiset(unordered_multiset&& other, const Allocator& a);
```

Construct a container moving `other`'s contained elements, and having the hash function, predicate and maximum load factor, but using allocate `a`.

**Notes:** This is implemented using Boost.Move.

**Requires:** `value_type` is move insertable.

---

## Initializer List Constructor

C++

```
unordered_multiset(std::initializer_list<value_type> il,
                   size_type n = implementation-defined,
                   const hasher& hf = hasher(),
                   const key_equal& eql = key_equal(),
                   const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

C++

```
unordered_multiset(size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate, `a` as the allocator and a maximum load factor of 1.0.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

```
unordered_multiset(size_type n, const hasher& hf, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate, `a` as the allocator and a maximum load factor of `1.0`.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` as the allocator, with the default hash function and key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate and a maximum load factor of `1.0` and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

```
unordered_multiset(std::initializer_list<value_type> il, const allocator_type& a);
```

C++

Constructs an empty container using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Allocator

C++

```
unordered_multiset(std::initializer_list<value_type> il, size_type n, const allocator_type& a)
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Hasher and Allocator

C++

```
unordered_multiset(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                   const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and a maximum load factor of 1.0 and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Destructor

C++

```
~unordered_multiset();
```

**Note:** The destructor is applied to every element, and all memory is deallocated

---

### Assignment

#### Copy Assignment

C++

```
unordered_multiset& operator=(const unordered_multiset& other);
```

The assignment operator Copies the contained elements, hash function, predicate and maximum load factor but not the allocator.

If `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, the allocator is overwritten, if not the copied elements are created using the existing allocator.

**Requires:** `value_type` is copy constructible

---

## Move Assignment

```
unordered_multiset& operator=(unordered_multiset&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&
        boost::is_nothrow_move_assignable_v<Hash> &&
        boost::is_nothrow_move_assignable_v<Pred>);
```

C++

The move assignment operator.

If `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`, the allocator is overwritten, if not the moved elements are created using the existing allocator.

**Requires:** `value_type` is move constructible.

---

## Initializer List Assignment

```
unordered_multiset& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All existing elements are either overwritten by the new elements or destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container and [CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable)).

---

## Iterators

### begin

```
iterator      begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

### end

```
iterator      end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

## cbegin

```
const_iterator cbegin() const noexcept;
```

C++

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

---

## cend

```
const_iterator cend() const noexcept;
```

C++

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

C++

**Returns:** `size() == 0`

---

### size

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

C++

**Returns:** `std::distance(begin(), end())`

---

### max\_size

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

C++

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

```
template<class... Args> iterator emplace(Args&&... args);
```

C++

Inserts an object, constructed with the arguments `args`, in the container.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## emplace\_hint

`template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);`

C++

Inserts an object, constructed with the arguments `args`, in the container.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `args`.

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Copy Insert

`iterator insert(const value_type& obj);`

C++

Inserts `obj` in the container.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Move Insert

iterator **insert**(value\_type&& obj);

C++

Inserts `obj` in the container.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Copy Insert with Hint

iterator **insert**(const\_iterator hint, const value\_type& obj);

C++

Inserts `obj` in the container.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Move Insert with Hint

```
iterator insert(const_iterator hint, value_type&& obj);
```

Inserts `obj` in the container.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** An iterator pointing to the inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Insert Iterator Range

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

Inserts a range of elements into the container.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into `X` from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

## Insert Initializer List

```
void insert(std::initializer_list<value_type> il);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

---

## Extract by Iterator

```
node_type extract(const_iterator position);
```

C++

Removes the element pointed to by `position`.

**Returns:** A `node_type` owning the element.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_set`.

---

## Extract by Value

```
node_type extract(const key_type& k);
template<class K> node_type extract(K&& k);
```

C++

Removes an element with key equivalent to `k`.

**Returns:** A `node_type` owning the element if found, otherwise an empty `node_type`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** A node extracted using this method can be inserted into a compatible `unordered_set`.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert with `node_handle`

```
iterator insert(node_type&& nh);
```

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh`.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns `end()`.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_set`.

---

### Insert with Hint and `node_handle`

iterator **insert**(const\_iterator hint, node\_type&& nh);

C++

If `nh` is empty, has no effect.

Otherwise inserts the element owned by `nh`.

`hint` is a suggestion to where the element should be inserted.

**Requires:** `nh` is empty or `nh.get_allocator()` is equal to the container's allocator.

**Returns:** If `nh` was empty, returns `end()`.

Otherwise returns an iterator pointing to the newly inserted element.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** The standard is fairly vague on the meaning of the hint. But the only practical way to use it, and the only way that Boost.Unordered supports is to point to an existing element with the same key.

Can invalidate iterators, but only if the insert causes the load factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

This can be used to insert a node extracted from a compatible `unordered_set`.

---

### Erase by Position

iterator **erase**(iterator position);  
iterator **erase**(const\_iterator position);

C++

Erase the element pointed to by `position`.

**Returns:** The iterator following `position` before the erasure.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** In older versions this could be inefficient because it had to search through several buckets to find the position of the returned iterator. The data structure has been changed so that this is no longer the case, and the alternative erase methods have been deprecated.

---

## Erase by Value

```
size_type erase(const key_type& k);  
template<class K> size_type erase(K&& x);
```

C++

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Erase Range

```
iterator erase(const_iterator first, const_iterator last);
```

C++

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

---

## quick\_erase

```
void quick_erase(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## erase\_return\_void

```
void erase_return_void(const_iterator position);
```

C++

Erase the element pointed to by `position`.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

In this implementation, this overload doesn't call either function object's methods so it is no throw, but this might not be true in other implementations.

**Notes:** This method was implemented because returning an iterator to the next element from `erase` was expensive, but the container has been redesigned so that is no longer the case. So this method is now deprecated.

---

## swap

```
void swap(unordered_multiset& noexcept(boost::allocator_traits<Allocator>::is_always_equal::value &&  
        boost::is_nothrow_swappable_v<Hash> &&  
        boost::is_nothrow_swappable_v<Pred>);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`

**Throws:** Never throws an exception.

---

## merge

```
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);
```

C++

Attempt to "merge" two containers by iterating `source` and extracting all nodes in `source` and inserting them into `*this`.

Because `source` can have a different hash function and key equality predicate, the key of each node in `source` is rehashed using `this->hash_function()` and then, if required, compared using `this->key_eq()`.

The behavior of this function is undefined if `this->get_allocator() != source.get_allocator()`.

This function does not copy or move any elements and instead simply relocates the nodes from `source` into `*this`.

**Notes:**

- Pointers and references to transferred elements remain valid.
- Invalidates iterators to transferred elements.
- Invalidates iterators belonging to `*this`.
- Iterators to non-transferred elements in `source` remain valid.

---

## Observers

### get\_allocator

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

C++

### hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

### key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<typename ComparableKey, typename ComparableHash, typename ComparablePredicate>
    iterator      find(ComparableKey const&, ComparableHash const&,
                           ComparablePredicate const&);
template<typename ComparableKey, typename ComparableHash, typename ComparablePredicate>
    const_iterator find(ComparableKey const&, ComparableHash const&,
                           ComparablePredicate const&) const;
```

C++

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.

**Notes:** The templated overloads containing `ComparableKey`, `ComparableHash` and `ComparablePredicate` are non-standard extensions which allow you to use a compatible hash function and equality predicate for a key of a different type in order to avoid an expensive type cast. In general, its use is not encouraged and instead the `K` member function templates should be used.

The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## count

```
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```
bool contains(const key_type& k) const;
template<class K>
bool contains(const K& k) const;
```

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

```
std::pair<iterator, iterator> equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
std::pair<iterator, iterator> equal_range(const K& k);
template<class K>
std::pair<const_iterator, const_iterator> equal_range(const K& k) const;
```

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

**Returns:** The number of buckets.

### max\_bucket\_count

```
size_type max_bucket_count() const noexcept;
```

**Returns:** An upper bound on the number of buckets.

### bucket\_size

```
size_type bucket_size(size_type n) const;
```

**Requires:**  $n < \text{bucket\_count}()$

**Returns:** The number of elements in bucket  $n$ .

---

## bucket

```
size_type bucket(const key_type& k) const;
template<class K> size_type bucket(const K& k) const;
```

**Returns:** The index of the bucket which would contain an element with key  $k$ .

**Postconditions:** The return value is less than  $\text{bucket\_count}()$ .

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## begin

```
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
```

**Requires:**  $n$  shall be in the range  $[0, \text{bucket\_count}())$ .

**Returns:** A local iterator pointing the first element in the bucket with index  $n$ .

---

## end

```
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
```

**Requires:**  $n$  shall be in the range  $[0, \text{bucket\_count}())$ .

**Returns:** A local iterator pointing the 'one past the end' element in the bucket with index  $n$ .

---

## cbegin

```
const_local_iterator cbegin(size_type n) const;
```

**Requires:**  $n$  shall be in the range  $[0, \text{bucket\_count}())$ .

**Returns:** A constant local iterator pointing the first element in the bucket with index `n`.

---

## cend

`const_local_iterator cend(size_type n) const;`

C++

**Requires:** `n` shall be in the range `[0, bucket_count())`.

**Returns:** A constant local iterator pointing the 'one past the end' element in the bucket with index `n`.

---

## Hash Policy

### load\_factor

`float load_factor() const noexcept;`

C++

**Returns:** The average number of elements per bucket.

---

### max\_load\_factor

`float max_load_factor() const noexcept;`

C++

**Returns:** Returns the current maximum load factor.

---

### Set max\_load\_factor

`void max_load_factor(float z);`

C++

**Effects:** Changes the container's maximum load factor, using `z` as a hint.

---

## rehash

`void rehash(size_type n);`

C++

Changes the number of buckets so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## reserve

```
void reserve(size_type n);
```

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`, or `a.rehash(1)` if `n > 0` and `a.max_load_factor() == std::numeric_limits<float>::infinity()`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, and changes the order of elements. Pointers and references to elements are not invalidated.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

## iter-value-type

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

## Equality Comparisons

### operator==

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& x,
                  const unordered_multiset<Key, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## operator!=

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& x,
                  const unordered_multiset<Key, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## Swap

```
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
          unordered_multiset<Key, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

**Notes:** The exception specifications aren't quite the same as the C++11 standard, as the equality predicate and hash function are swapped using their copy constructors.

## erase\_if

```
template<class K, class H, class P, class A, class Predicate>
typename unordered_multiset<K, H, P, A>::size_type
erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
C++
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

## Serialization

`unordered_multiset` s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_multiset` to an archive

Saves all the elements of an `unordered_multiset` `x` to an archive (XML archive) `ar`.

**Requires:** `value_type` is serializable (XML serializable), and it supports Boost.Serialization `save_construct_data / load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

### Loading an `unordered_multiset` from an archive

Deletes all preexisting elements of an `unordered_multiset` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_multiset` `other` saved to the storage read by `ar`.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)). `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Note:** If the archive was saved using a release of Boost prior to Boost 1.84, the configuration macro `BOOST_UNORDERED_ENABLE_SERIALIZATION_COMPATIBILITY_V0` has to be globally defined for this operation to succeed; otherwise, an exception is thrown.

---

### Saving an iterator/`const_iterator` to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be an `end()` iterator.

**Requires:** The `unordered_multiset` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

### Loading an iterator/`const_iterator` from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_multiset` it points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Hash traits

### Synopsis

```
// #include <boost/unordered/hash_traits.hpp>

namespace boost {
namespace unordered {

template<typename Hash>
struct hash_is_avalanching;

} // namespace unordered
} // namespace boost
```

---

### hash\_is\_avalanching

```
template<typename Hash>
struct hash_is_avalanching;
```

C++

A hash function is said to have the *avalanching property* if small changes in the input translate to large changes in the returned hash code —ideally, flipping one bit in the representation of the input value results in each bit of the hash code flipping with probability 50%. Approaching this property is critical for the proper behavior of open-addressing hash containers.

`hash_is_avalanching<Hash>::value` is `true` if `Hash::is_avalanching` is a valid type, and `false` otherwise. Users can then declare a hash function `Hash` as avalanching either by embedding an `is_avalanching` `typedef` into the definition of `Hash`, or directly by specializing `hash_is_avalanching<Hash>` to a class with an embedded compile-time constant `value` set to `true`.

Open-addressing and concurrent containers use the provided hash function `Hash` as-is if

`hash_is_avalanching<Hash>::value` is `true`; otherwise, they implement a bit-mixing post-processing stage to increase the quality of hashing at the expense of extra computational cost.

---

## Class Template unordered\_flat\_map

`boost::unordered_flat_map` — An open-addressing unordered associative container that associates unique keys with another value.

The performance of `boost::unordered_flat_map` is much better than that of `boost::unordered_map` or other implementations of `std::unordered_map`. Unlike standard unordered associative containers, which are node-based, the elements of a `boost::unordered_flat_map` are held directly in the bucket array, and insertions into an already occupied bucket are diverted to available buckets in the vicinity of the original position. This type of data layout is known as *open addressing*.

As a result of its using open addressing, the interface of `boost::unordered_flat_map` deviates in a number of aspects from that of `boost::unordered_flat_map / std::unordered_flat_map`:

- `value_type` must be move-constructible.
- Pointer stability is not kept under rehashing.
- `begin()` is not constant-time.
- There is no API for bucket handling (except `bucket_count`) or node extraction/insertion.
- The maximum load factor of the container is managed internally and can't be set by the user.

Other than this, `boost::unordered_flat_map` is mostly a drop-in replacement of node-based standard unordered associative containers.

## Synopsis



```

unordered_flat_map(std::initializer_list<value_type> il, const allocator_type& a);
unordered_flat_map(std::initializer_list<value_type> il, size_type n,
                  const allocator_type& a);
unordered_flat_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                  const allocator_type& a);
~unordered_flat_map();
unordered_flat_map& operator=(const unordered_flat_map& other);
unordered_flat_map& operator=(unordered_flat_map&& other) noexcept(
    boost::allocator_traits<Allocator>::is_always_equal::value || 
    boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
    std::is_same<pointer, value_type*>::value);
unordered_flat_map& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(const init_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
std::pair<iterator, bool> insert(init_type&& obj);
iterator      insert(const_iterator hint, const value_type& obj);
iterator      insert(const_iterator hint, const init_type& obj);
iterator      insert(const_iterator hint, value_type&& obj);
iterator      insert(const_iterator hint, init_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type>);

template<class... Args>
    std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
    std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
    std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
    iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
template<class M>
    std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
    std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
    std::pair<iterator, bool> insert_or_assign(const_iterator hint, K&& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

```

```

template<class K, class M>
iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);

convertible-to-iterator      erase(iterator position);
convertible-to-iterator      erase(const_iterator position);
size_type                   erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator       erase(const_iterator first, const_iterator last);
void          swap(unordered_flat_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
              boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
void          clear() noexcept;

template<class H2, class P2>
void merge(unordered_flat_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_flat_map<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
template<class K>
const_iterator find(const K& k) const;
size_type     count(const key_type& k) const;
template<class K>
size_type     count(const K& k) const;
bool         contains(const key_type& k) const;
template<class K>
bool         contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>  equal_range(const key_type& k) const;
template<class K>
std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
std::pair<const_iterator, const_iterator>  equal_range(const K& k) const;

// element access
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

```

```

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-key-type<InputIterator>>,
         class Pred = std::equal_to<iter-key-type<InputIterator>>,
         class Allocator = std::allocator<iter-to-alloc-type<InputIterator>>>
unordered_flat_map(InputIterator, InputIterator, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  Pred, Allocator>;

template<class Key, class T, class Hash = boost::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
unordered_flat_map(std::initializer_list<std::pair<Key, T>>,
                  typename see below::size_type = see below, Hash = Hash(),
                  Pred = Pred(), Allocator = Allocator())
-> unordered_flat_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_flat_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_flat_map(InputIterator, InputIterator, Allocator)
-> unordered_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_flat_map(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_flat_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Allocator)
-> unordered_flat_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_flat_map(std::initializer_list<std::pair<Key, T>>, Allocator)
-> unordered_flat_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Hash, class Allocator>
unordered_flat_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Hash, Allocator)
-> unordered_flat_map<Key, T, Hash, std::equal_to<Key>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_map<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_map<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_flat_map<Key, T, Hash, Pred, Alloc>& y)

```

```

noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_flat_map<K, T, H, P, A>::size_type
    erase_if(unordered_flat_map<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<i>Key</i> and <i>T</i> must be <a href="#">MoveConstructible</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/MoveConstructible">https://en.cppreference.com/w/cpp/named_req/MoveConstructible</a> ). <i>std::pair&lt;const Key, T&gt;</i> must be <a href="#">EmplaceConstructible</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible">https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible</a> ) into the container from any <i>std::pair</i> object convertible to it, and it also must be <a href="#">Erasable</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container.
<i>T</i>	
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <i>std::size_t</i> .
<i>Pred</i>	A binary function object that induces an equivalence relation on values of type <i>Key</i> . It takes two arguments of type <i>Key</i> and returns a value of type <i>bool</i> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <a href="#">fancy pointers</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The elements of the container are held into an internal *bucket array*. An element is inserted into a bucket determined by its hash code, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert` / `emplace`, or as a result of calling `rehash` / `reserve`. The *load factor* of the container (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

## Typedefs

C++

```
typedef implementation-defined iterator;
```

An iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

```
typedef implementation-defined const_iterator;
```

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

## Constructors

### Default Constructor

```
unordered_flat_map();
```

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_flat_map(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Iterator Range Constructor

```
template<class InputIterator>
unordered_flat_map(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

```
unordered_flat_map(unordered_flat_map const& other);
```

C++

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

---

## Move Constructor

```
unordered_flat_map(unordered_flat_map&& other);
```

C++

The move constructor. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

---

## Iterator Range Constructor with Allocator

```
template<class InputIterator>
unordered_flat_map(InputIterator f, InputIterator l, const allocator_type& a);
```

C++

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

```
explicit unordered_flat_map(Allocator const& a);
```

C++

Constructs an empty container, using allocator `a`.

---

## Copy Constructor with Allocator

```
unordered_flat_map(unordered_flat_map const& other, Allocator const& a);
```

C++

Constructs a container, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

---

## Move Constructor with Allocator

```
unordered_flat_map(unordered_flat_map&& other, Allocator const& a);
```

C++

If `a == other.get_allocator()`, the elements of `other` are transferred directly to the new container; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

---

## Move Constructor from concurrent\_flat\_map

```
unordered_flat_map(concurrent_flat_map<Key, T, Hash, Pred, Allocator>&& other);
```

C++

Move construction from a `concurrent_flat_map`. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

**Complexity:** Constant time.

**Concurrency:** Blocking on `other`.

---

## Initializer List Constructor

```
unordered_flat_map(std::initializer_list<value_type> il,
                   size_type n = implementation-defined
                   const hasher& hf = hasher(),
                   const key_equal& eql = key_equal(),
                   const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

```
unordered_flat_map(size_type n, allocator_type const& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

C++

```
unordered_flat_map(size_type n, hasher const& hf, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

C++

```
template<class InputIterator>
unordered_flat_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

C++

```
template<class InputIterator>
unordered_flat_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

C++

```
unordered_flat_map(std::initializer_list<value_type> il, const allocator_type& a);
```

Constructs an empty container using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Allocator

C++

```
unordered_flat_map(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

Constructs an empty container with at least n buckets, using a and default hash function and key equality predicate, and inserts the elements from il into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Hasher and Allocator

C++

```
unordered_flat_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                   const allocator_type& a);
```

Constructs an empty container with at least n buckets, using hf as the hash function, a as the allocator and default key equality predicate, and inserts the elements from il into it.

**Requires:** key\_equal needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Destructor

C++

```
~unordered_flat_map();
```

**Note:** The destructor is applied to every element, and all memory is deallocated

---

### Assignment

#### Copy Assignment

C++

```
unordered_flat_map& operator=(unordered_flat_map const& other);
```

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from other , copy-assigns the allocator from other if Alloc::propagate\_on\_container\_copy\_assignment exists and Alloc::propagate\_on\_container\_copy\_assignment::value is true , and finally inserts copies of the elements of other .

**Requires:** value\_type is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

## Move Assignment

```
unordered_flat_map& operator=(unordered_flat_map&& other)
    noexcept((boost::allocator_traits<Allocator>::is_always_equal::value ||
        boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
        std::is_same<pointer, value_type*>::value);
```

C++

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to the new container; otherwise, inserts move-constructed copies of the elements of `other`.

---

## Initializer List Assignment

```
unordered_flat_map& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

## Iterators

### begin

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:** O(bucket\_count())

---

### end

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

---

### cbegin

```
const_iterator cbegin() const noexcept;
```

C++

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:** O(bucket\_count())

---

## cend

```
const_iterator cend() const noexcept;
```

C++

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

C++

**Returns:** `size() == 0`

---

### size

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

C++

**Returns:** `std::distance(begin(), end())`

---

### max\_size

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

C++

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

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

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## emplace\_hint

```
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
```

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

`position` is a suggestion to where the element should be inserted. This implementation ignores it.

- Requires:** `value_type` is constructible from `args`.
- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Copy Insert

```
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(const init_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

- Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).
- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(x)`, where `x` is equally convertible to both `const value_type&` and `const init_type&`, is not ambiguous and selects the `init_type` overload.

---

## Move Insert

```
std::pair<iterator, bool> insert(value_type&& obj);
std::pair<iterator, bool> insert(init_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(x)`, where `x` is equally convertible to both `value_type&&` and `init_type&&`, is not ambiguous and selects the `init_type` overload.

---

## Copy Insert with Hint

```
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, const init_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(hint, x)`, where `x` is equally convertible to both `const value_type&` and `const init_type&`, is not ambiguous and selects the `init_type` overload.

---

## Move Insert with Hint

```
iterator insert(const_iterator hint, value_type&& obj);  
iterator insert(const_iterator hint, init_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(hint, x)`, where `x` is equally convertible to both `value_type&&` and `init_type&&`, is not ambiguous and selects the `init_type` overload.

---

## Insert Iterator Range

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into the container from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Initializer List

C++

```
void insert(std::initializer_list<value_type>);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## try\_emplace

C++

```
template<class... Args>
std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
```

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similar to emplace, with the difference that no `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

C++

unlike `emplace`, which simply forwards all arguments to `value_type`'s constructor.

Can invalidate iterators pointers and references, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## try\_emplace with Hint

```
template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
```

C++

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similar to `emplace_hint`, with the difference that no `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

C++

unlike `emplace_hint`, which simply forwards all arguments to `value_type`'s constructor.

Can invalidate iterators pointers and references, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## insert\_or\_assign

```
template<class M>
std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
std::pair<iterator, bool> insert_or_assign(K&& k, M&& obj);
```

C++

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

C++

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators pointers and references, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## insert\_or\_assign with Hint

C++

```
template<class M>
iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
template<class K, class M>
iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);
```

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

C++

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

The template<class K, class M> only participates in overload resolution if Hash::is\_transparent and Pred::is\_transparent are valid member typedefs. The library assumes that Hash is callable with both K and Key and that Pred is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the Key type.

---

## Erase by Position

C++

```
convertible-to-iterator erase(iterator position);
convertible-to-iterator erase(const_iterator position);
```

Erase the element pointed to by `position`.

**Returns:** An opaque object implicitly convertible to the `iterator` or `const_iterator` immediately following `position` prior to the erasure.

**Throws:** Nothing.

**Notes:** The opaque object returned must only be discarded or immediately converted to `iterator` or `const_iterator`.

---

## Erase by Key

C++

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The template<class K> overload only participates in overload resolution if Hash::is\_transparent and Pred::is\_transparent are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from K. The library assumes that Hash is callable with both K and Key and that Pred is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the Key type.

---

## Erase Range

C++

```
iterator erase(const_iterator first, const_iterator last);
```

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Nothing in this implementation (neither the `hasher` nor the `key_equal` objects are called).

---

## swap

```
void swap(unordered_flat_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

---

## merge

```
template<class H2, class P2>
void merge(unordered_flat_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_flat_map<Key, T, H2, P2, Allocator>&& source);
```

C++

Move-inserts all the elements from `source` whose key is not already present in `*this`, and erases them from `source`.

---

## Observers

### get\_allocator

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

C++

**Returns:** The container's allocator.

---

## hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

## key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
```

C++

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `end()` if no such element exists.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

### count

```
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

### contains

```
bool      contains(const key_type& k) const;
template<class K>
bool      contains(const K& k) const;
```

C++

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

```
std::pair<iterator, iterator>           equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>           equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;
```

C++

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## operator[]

```
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);
```

C++

**Effects:** If the container does not already contain an element with a key equivalent to `k`, inserts the value `std::pair<key_type const, mapped_type>(k, mapped_type())`.

**Returns:** A reference to `x.second` where `x` is the element already in the container, or the newly inserted element with a key equivalent to `k`.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

at

C++

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;
```

**Returns:** A reference to `x.second` where `x` is the (unique) element whose key is equivalent to `k`.

**Throws:** An exception object of type `std::out_of_range` if no such element is present.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

`bucket_count`

C++

```
size_type bucket_count() const noexcept;
```

**Returns:** The size of the bucket array.

---

## Hash Policy

`load_factor`

C++

```
float load_factor() const noexcept;
```

**Returns:** `static_cast<float>(size()) / static_cast<float>(bucket_count())`, or 0 if `bucket_count() == 0`.

---

`max_load_factor`

C++

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the container's maximum load factor.

---

Set `max_load_factor`

C++

```
void max_load_factor(float z);
```

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with `boost::unordered_map`.

---

## max\_load

`size_type max_load() const noexcept;`

C++

**Returns:** The maximum number of elements the container can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the container's maximum load is at least `max_load_factor() * bucket_count()`. This number may decrease on erasure under high-load conditions.

---

## rehash

`void rehash(size_type n);`

C++

Changes if necessary the size of the bucket array so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array. If the provided Allocator uses fancy pointers, a default allocation is subsequently performed.

Invalidates iterators, pointers and references, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## reserve

`void reserve(size_type n);`

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, pointers and references, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
  - It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.
- A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

### *iter-key-type*

```
template<class InputIterator>
using iter-key-type = std::remove_const_t<
    std::tuple_element_t<0, iter-value-type<InputIterator>>; // exposition only
```

### *iter-mapped-type*

```
template<class InputIterator>
using iter-mapped-type =
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

### *iter-to-alloc-type*

```
template<class InputIterator>
using iter-to-alloc-type = std::pair<
    std::add_const_t<std::tuple_element_t<0, iter-value-type<InputIterator>>,
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

## Equality Comparisons

### operator==

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

### operator!=

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_flat_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_flat_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y))));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_flat_map<K, T, H, P, A>::size_type
erase_if(unordered_flat_map<K, T, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

C++

## Serialization

`unordered_flat_map`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

## Saving an `unordered_flat_map` to an archive

Saves all the elements of an `unordered_flat_map` `x` to an archive (XML archive) `ar`.

**Requires:** `std::remove_const<key_type>::type` and `std::remove_const<mapped_type>::type` are serializable (XML serializable), and they do support Boost.Serialization `save_construct_data` / `load_construct_data` protocol (automatically supported by DefaultConstructible ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

### Loading an `unordered_flat_map` from an archive

Deletes all preexisting elements of an `unordered_flat_map` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_flat_map` `other` saved to the storage read by `ar`.

**Requires:** `x.key_equal()` is functionally equivalent to `other.key_equal()`.

---

### Saving an iterator/`const_iterator` to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be and `end()` iterator.

**Requires:** The `unordered_flat_map` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

### Loading an iterator/`const_iterator` from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_flat_map` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

---

## Class Template `unordered_flat_set`

`boost::unordered_flat_set` — An open-addressing unordered associative container that stores unique values.

The performance of `boost::unordered_flat_set` is much better than that of `boost::unordered_set` or other implementations of `std::unordered_set`. Unlike standard unordered associative containers, which are node-based, the elements of a `boost::unordered_flat_set` are held directly in the bucket array, and insertions into an already occupied bucket are diverted to available buckets in the vicinity of the original position. This type of data layout is known as *open addressing*.

As a result of its using open addressing, the interface of `boost::unordered_flat_set` deviates in a number of aspects from that of `boost::unordered_flat_set` / `std::unordered_flat_set`:

- `value_type` must be move-constructible.
- Pointer stability is not kept under rehashing.
- `begin()` is not constant-time.

- There is no API for bucket handling (except `bucket_count`) or node extraction/insertion.
  - The maximum load factor of the container is managed internally and can't be set by the user.
- Other than this, `boost::unordered_flat_set` is mostly a drop-in replacement of node-based standard unordered associative containers.

## Synopsis

```

// #include <boost/unordered/unordered_flat_set.hpp>

namespace boost {
    template<class Key,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<Key>>
    class unordered_flat_set {
public:
    // types
    using key_type          = Key;
    using value_type         = Key;
    using init_type          = Key;
    using hasher             = Hash;
    using key_equal          = Pred;
    using allocator_type     = Allocator;
    using pointer             = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer       = typename std::allocator_traits<Allocator>::const_pointer;
    using reference           = value_type&;
    using const_reference     = const value_type&;
    using size_type           = std::size_t;
    using difference_type     = std::ptrdiff_t;

    using iterator            = implementation-defined;
    using const_iterator       = implementation-defined;

    // construct/copy/destroy
    unordered_flat_set();
    explicit unordered_flat_set(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_flat_set(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_flat_set(const unordered_flat_set& other);
    unordered_flat_set(unordered_flat_set&& other);
    template<class InputIterator>
    unordered_flat_set(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_flat_set(const Allocator& a);
    unordered_flat_set(const unordered_flat_set& other, const Allocator& a);
    unordered_flat_set(concurrent_flat_set<Key, Hash, Pred, Allocator>&& other);
    unordered_flat_set(std::initializer_list<value_type> il,
                      size_type n = implementation-defined
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_flat_set(size_type n, const allocator_type& a);
    unordered_flat_set(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_flat_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
    unordered_flat_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
    unordered_flat_set(std::initializer_list<value_type> il, const allocator_type& a);
    unordered_flat_set(std::initializer_list<value_type> il, size_type n,
                      const allocator_type& a);
    unordered_flat_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                      const allocator_type& a);
~unordered_flat_set();
}

```

```

unordered_flat_set& operator=(const unordered_flat_set& other);
unordered_flat_set& operator=(unordered_flat_set&& other) noexcept(
    boost::allocator_traits<Allocator>::is_always_equal::value || 
    boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
    std::is_same<pointer, value_type*>::value);
unordered_flat_set& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
template<class K> std::pair<iterator, bool> insert(K&& k);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class K> iterator insert(const_iterator hint, K&& k);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type>);

convertible-to-iterator      erase(iterator position);
convertible-to-iterator      erase(const_iterator position);
size_type                   erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator      erase(const_iterator first, const_iterator last);
void         swap(unordered_flat_set& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value || 
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
void         clear() noexcept;

template<class H2, class P2>
void merge(unordered_flat_set<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_flat_set<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
template<class K>
const_iterator find(const K& k) const;
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
bool          contains(const key_type& k) const;

```

```

template<class K>
    bool contains(const K& k) const;
std::pair<iterator, iterator> equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator> equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-value-type<InputIterator>>,
         class Pred = std::equal_to<iter-value-type<InputIterator>>,
         class Allocator = std::allocator<iter-value-type<InputIterator>>>
unordered_flat_set(InputIterator, InputIterator, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_flat_set<iter-value-type<InputIterator>, Hash, Pred, Allocator>;

template<class T, class Hash = boost::hash<T>, class Pred = std::equal_to<T>,
         class Allocator = std::allocator<T>>
unordered_flat_set(std::initializer_list<T>, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_flat_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_flat_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_flat_set<iter-value-type<InputIterator>,
               boost::hash<iter-value-type<InputIterator>>,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_flat_set(InputIterator, InputIterator, Allocator)
-> unordered_flat_set<iter-value-type<InputIterator>,
               boost::hash<iter-value-type<InputIterator>>,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_flat_set(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_flat_set<iter-value-type<InputIterator>, Hash,
                  std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class T, class Allocator>
unordered_flat_set(std::initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_flat_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Allocator>
unordered_flat_set(std::initializer_list<T>, Allocator)
-> unordered_flat_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>

```

```

unordered_flat_set(std::initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_flat_set<T, Hash, std::equal_to<T>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_set<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_set<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
          unordered_flat_set<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_flat_set<K, T, H, P, A>::size_type
erase_if(unordered_flat_set<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<i>Key</i> must be <u><a href="#">MoveInsertable</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/MoveInsertable">https://en.cppreference.com/w/cpp/named_req/MoveInsertable</a> ) into the container and <u><a href="#">Erasable</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container.
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that induces an equivalence relation on values of type <i>Key</i> . It takes two arguments of type <i>Key</i> and returns a value of type <code>bool</code> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <u><a href="#">fancy pointers</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The elements of the container are held into an internal *bucket array*. An element is inserted into a bucket determined by its hash code, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert` / `emplace`, or as a result of calling `rehash` / `reserve`. The *load factor* of the container (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

---

## Typedefs

C++

```
typedef implementation-defined iterator;
```

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

---

C++

```
typedef implementation-defined const_iterator;
```

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

## Constructors

### Default Constructor

C++

```
unordered_flat_set();
```

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

C++

```
explicit unordered_flat_set(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Iterator Range Constructor

C++

```
template<class InputIterator>
unordered_flat_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Copy Constructor

C++

```
unordered_flat_set(unordered_flat_set const& other);
```

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

## Move Constructor

C++

```
unordered_flat_set(unordered_flat_set&& other);
```

The move constructor. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

## Iterator Range Constructor with Allocator

C++

```
template<class InputIterator>
unordered_flat_set(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Allocator Constructor

```
explicit unordered_flat_set(Allocator const& a);
```

C++

Constructs an empty container, using allocator `a`.

---

### Copy Constructor with Allocator

```
unordered_flat_set(unordered_flat_set const& other, Allocator const& a);
```

C++

Constructs a container, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

---

### Move Constructor with Allocator

```
unordered_flat_set(unordered_flat_set&& other, Allocator const& a);
```

C++

If `a == other.get_allocator()`, the elements of `other` are transferred directly to the new container; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

---

### Move Constructor from concurrent\_flat\_set

```
unordered_flat_set(concurrent_flat_set<Key, Hash, Pred, Allocator>&& other);
```

C++

Move construction from a `concurrent_flat_set`. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

**Complexity:** Constant time.

**Concurrency:** Blocking on `other`.

---

### Initializer List Constructor

```
unordered_flat_set(std::initializer_list<value_type> il,
                  size_type n = implementation-defined
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor with Allocator

```
unordered_flat_set(size_type n, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

```
unordered_flat_set(size_type n, hasher const& hf, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
unordered_flat_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>
unordered_flat_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

```
unordered_flat_set(std::initializer_list<value_type> il, const allocator_type& a);
```

C++

Constructs an empty container using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Allocator

```
unordered_flat_set(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

```
unordered_flat_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,  
                  const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and default key equality predicate, and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Destructor

```
~unordered_flat_set();
```

C++

**Note:** The destructor is applied to every element, and all memory is deallocated

---

## Assignment

### Copy Assignment

```
unordered_flat_set& operator=(unordered_flat_set const& other);
```

C++

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from `other`, copy-assigns the allocator from `other` if `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, and finally inserts copies of the elements of `other`.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

---

## Move Assignment

```
unordered_flat_set& operator=(unordered_flat_set&& other)
    noexcept((boost::allocator_traits<Allocator>::is_always_equal::value ||
        boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
        std::is_same<pointer, value_type*>::value);
```

C++

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to the new container; otherwise, inserts move-constructed copies of the elements of `other`.

---

## Initializer List Assignment

```
unordered_flat_set& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

## Iterators

### begin

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:** O(bucket\_count())

---

### end

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

---

## cbegin

```
const_iterator cbegin() const noexcept;
```

C++

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:** O(bucket\_count())

---

## cend

```
const_iterator cend() const noexcept;
```

C++

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

C++

**Returns:** size() == 0

---

### size

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

C++

**Returns:** std::distance(begin(), end())

---

### max\_size

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

C++

**Returns:** size() of the largest possible container.

---

## Modifiers

### emplace

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

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## emplace\_hint

```
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
```

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

`position` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Copy Insert

```
std::pair<iterator, bool> insert(const value_type& obj);
```

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Move Insert

```
std::pair<iterator, bool> insert(value_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

- Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).
- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.
- Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Transparent Insert

```
template<class K> std::pair<iterator, bool> insert(K&& k);
```

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

- Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.
- Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.
- Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Copy Insert with Hint

```
iterator insert(const_iterator hint, const value_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Move Insert with Hint

```
iterator insert(const_iterator hint, value_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Transparent Insert with Hint

```
template<class K> std::pair<iterator, bool> insert(const_iterator hint, K&& k);
```

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.

**Returns:** The `bool` component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert Iterator Range

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into the container from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Initializer List

C++

```
void insert(std::initializer_list<value_type>);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, pointers and references, but only if the insert causes the load to be greater than the maximum load.

---

## Erase by Position

C++

```
convertible-to-iterator erase(iterator position);
convertible-to-iterator erase(const_iterator position);
```

Erase the element pointed to by `position`.

**Returns:** An opaque object implicitly convertible to the `iterator` or `const_iterator` immediately following `position` prior to the erasure.

**Throws:** Nothing.

**Notes:** The opaque object returned must only be discarded or immediately converted to `iterator` or `const_iterator`.

---

## Erase by Key

C++

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## Erase Range

```
iterator erase(const_iterator first, const_iterator last);
```

C++

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Nothing in this implementation (neither the `hasher` nor the `key_equal` objects are called).

---

## swap

```
void swap(unordered_flat_set& other)  
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||  
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

---

## merge

```
template<class H2, class P2>  
void merge(unordered_flat_set<Key, T, H2, P2, Allocator>& source);  
template<class H2, class P2>  
void merge(unordered_flat_set<Key, T, H2, P2, Allocator>&& source);
```

C++

Move-inserts all the elements from `source` whose key is not already present in `*this`, and erases them from `source`.

---

## Observers

### get\_allocator

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

C++

**Returns:** The container's allocator.

---

## hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

## key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
```

C++

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `end()` if no such element exists.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## count

```
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```
bool contains(const key_type& k) const;
template<class K>
    bool contains(const K& k) const;
```

C++

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## equal\_range

```
std::pair<iterator, iterator> equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator> equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;
```

C++

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

C++

**Returns:** The size of the bucket array.

---

## Hash Policy

## load\_factor

C++

```
float load_factor() const noexcept;
```

**Returns:** static\_cast<float>(size()) / static\_cast<float>(bucket\_count()), or 0 if bucket\_count() == 0.

---

## max\_load\_factor

C++

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the container's maximum load factor.

---

## Set max\_load\_factor

C++

```
void max_load_factor(float z);
```

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with boost::unordered\_set .

---

## max\_load

C++

```
size_type max_load() const noexcept;
```

**Returns:** The maximum number of elements the container can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the container's maximum load is at least max\_load\_factor() \* bucket\_count() . This number may decrease on erasure under high-load conditions.

---

## rehash

C++

```
void rehash(size_type n);
```

Changes if necessary the size of the bucket array so that there are at least n buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the bucket\_count() associated with the container.

When size() == 0 , rehash(0) will deallocate the underlying buckets array. If the provided Allocator uses fancy pointers, a default allocation is subsequently performed.

Invalidates iterators, pointers and references, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## reserve

```
void reserve(size_type n);
```

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators, pointers and references, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

## *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

## Equality Comparisons

### `operator==`

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_set<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## operator!=

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_flat_set<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_flat_set<Key, T, Hash, Pred, Alloc>& x,
          unordered_flat_set<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_flat_set<K, T, H, P, A>::size_type
erase_if(unordered_flat_set<K, T, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
C++
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

## Serialization

`unordered_flat_set` s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_flat_set` to an archive

Saves all the elements of an `unordered_flat_set` `x` to an archive (XML archive) `ar`.

**Requires:** `value_type` is serializable (XML serializable), and it supports Boost.Serialization `save_construct_data / load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

### Loading an `unordered_flat_set` from an archive

Deletes all preexisting elements of an `unordered_flat_set` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_flat_set` `other` saved to the storage read by `ar`.

**Requires:** `x.key_equal()` is functionally equivalent to `other.key_equal()`.

---

### Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be an `end()` iterator.

**Requires:** The `unordered_flat_set` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

### Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_flat_set` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template `unordered_node_map`

`boost::unordered_node_map` — A node-based, open-addressing unordered associative container that associates unique keys with another value.

`boost::unordered_node_map` uses an open-addressing layout like `boost::unordered_flat_map`, but, being node-based, it provides pointer/iterator stability and node handling functionalities. Its performance lies between those of `boost::unordered_map` and `boost::unordered_flat_map`.

As a result of its using open addressing, the interface of `boost::unordered_node_map` deviates in a number of aspects from that of `boost::unordered_map` / `std::unordered_map`:

- `begin()` is not constant-time.
- There is no API for bucket handling (except `bucket_count`).
- The maximum load factor of the container is managed internally and can't be set by the user.

Other than this, `boost::unordered_node_map` is mostly a drop-in replacement of standard unordered associative containers.

## Synopsis

```

// #include <boost/unordered/unordered_node_map.hpp>

namespace boost {
    template<class Key,
              class T,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<std::pair<const Key, T>>>
    class unordered_node_map {
public:
    // types
    using key_type           = Key;
    using mapped_type         = T;
    using value_type          = std::pair<const Key, T>;
    using init_type           = std::pair<
        typename std::remove_const<Key>::type,
        typename std::remove_const<T>::type
    >;
    using hasher              = Hash;
    using key_equal           = Pred;
    using allocator_type       = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer         = typename std::allocator_traits<Allocator>::const_pointer;
    using reference             = value_type&;
    using const_reference       = const value_type&;
    using size_type             = std::size_t;
    using difference_type      = std::ptrdiff_t;

    using iterator             = implementation-defined;
    using const_iterator        = implementation-defined;

    using node_type             = implementation-defined;
    using insert_return_type     = implementation-defined;

    // construct/copy/destroy
    unordered_node_map();
    explicit unordered_node_map(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_node_map(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_node_map(const unordered_node_map& other);
    unordered_node_map(unordered_node_map&& other);
    template<class InputIterator>
    unordered_node_map(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_node_map(const Allocator& a);
    unordered_node_map(const unordered_node_map& other, const Allocator& a);
    unordered_node_map(unordered_node_map&& other, const Allocator& a);
    unordered_node_map(std::initializer_list<value_type> il,
                      size_type n = implementation-defined
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_node_map(size_type n, const allocator_type& a);
    unordered_node_map(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_node_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>

```

```

unordered_node_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                  const allocator_type& a);
unordered_node_map(std::initializer_list<value_type> il, const allocator_type& a);
unordered_node_map(std::initializer_list<value_type> il, size_type n,
                  const allocator_type& a);
unordered_node_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                  const allocator_type& a);
~unordered_node_map();
unordered_node_map& operator=(const unordered_node_map& other);
unordered_node_map& operator=(unordered_node_map&& other) noexcept(
    (boost::allocator_traits<Allocator>::is_always_equal::value ||
     boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
     std::is_same<pointer, value_type*>::value);
unordered_node_map& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(const init_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
std::pair<iterator, bool> insert(init_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, const init_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
iterator insert(const_iterator hint, init_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type>);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class... Args>
    std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
    std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
    std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
    iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
template<class M>
    std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
    std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
    std::pair<iterator, bool> insert_or_assign(K&& k, M&& obj);

```

```

template<class M>
    iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
template<class K, class M>
    iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);

convertible-to-iterator      erase(iterator position);
convertible-to-iterator      erase(const_iterator position);
size_type                   erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator      erase(const_iterator first, const_iterator last);
void         swap(unordered_node_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
node_type extract(const_iterator position);
node_type extract(const key_type& key);
template<class K> node_type extract(K&& key);
void         clear() noexcept;

template<class H2, class P2>
    void merge(unordered_node_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_node_map<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

// element access
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;

```

```

float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-key-type<InputIterator>>,
         class Pred = std::equal_to<iter-key-type<InputIterator>>,
         class Allocator = std::allocator<iter-to-alloc-type<InputIterator>>>
unordered_node_map(InputIterator, InputIterator, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_node_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  Pred, Allocator>;

template<class Key, class T, class Hash = boost::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
unordered_node_map(std::initializer_list<std::pair<Key, T>>,
                  typename see below::size_type = see below, Hash = Hash(),
                  Pred = Pred(), Allocator = Allocator())
-> unordered_node_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_node_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_node_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_node_map(InputIterator, InputIterator, Allocator)
-> unordered_node_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                  boost::hash<iter-key-type<InputIterator>>,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_node_map(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_node_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                  std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_node_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Allocator)
-> unordered_node_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_node_map(std::initializer_list<std::pair<Key, T>>, Allocator)
-> unordered_node_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Hash, class Allocator>
unordered_node_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                  Hash, Allocator)
-> unordered_node_map<Key, T, Hash, std::equal_to<Key>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_map<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>

```

```

bool operator!=(const unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_map<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_node_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_node_map<K, T, H, P, A>::size_type
erase_if(unordered_node_map<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<code>std::pair&lt;const Key, T&gt;</code> must be <a href="#">EmplaceConstructible</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible">https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible</a> ) into the container from any <code>std::pair</code> object convertible to it, and it also must be <a href="#">Erasable</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container.
<i>T</i>	
<i>Hash</i>	A unary function object type that acts a hash function for a <code>Key</code> . It takes a single argument of type <code>Key</code> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that induces an equivalence relation on values of type <code>Key</code> . It takes two arguments of type <code>Key</code> and returns a value of type <code>bool</code> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <a href="#">fancy pointers</a> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The element nodes of the container are held into an internal *bucket array*. A node is inserted into a bucket determined by the hash code of its element, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert` / `emplace`, or as a result of calling `rehash` / `reserve`. The *load factor* of the container (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

## Typedefs

```
typedef implementation-defined iterator;
```

C++

An iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

---

```
typedef implementation-defined const_iterator;
```

C++

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

---

```
typedef implementation-defined node_type;
```

C++

A class for holding extracted container elements, modelling [NodeHandle](#) ([https://en.cppreference.com/w/cpp/container/node\\_handle](https://en.cppreference.com/w/cpp/container/node_handle)).

---

```
typedef implementation-defined insert_return_type;
```

C++

A specialization of an internal class template:

```
template<class Iterator, class NodeType>
struct insert_return_type // name is exposition only
{
    Iterator position;
    bool inserted;
    NodeType node;
};
```

C++

with `Iterator = iterator` and `NodeType = node_type`.

---

## Constructors

### Default Constructor

```
unordered_node_map();
```

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor

C++

```
explicit unordered_node_map(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor

C++

```
template<class InputIterator>
unordered_node_map(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

C++

```
unordered_node_map(unordered_node_map const& other);
```

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

---

## Move Constructor

C++

```
unordered_node_map(unordered_node_map&& other);
```

The move constructor. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

---

## Iterator Range Constructor with Allocator

C++

```
template<class InputIterator>
unordered_node_map(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

C++

```
explicit unordered_node_map(Allocator const& a);
```

Constructs an empty container, using allocator `a`.

---

## Copy Constructor with Allocator

C++

```
unordered_node_map(unordered_node_map const& other, Allocator const& a);
```

Constructs a container, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

---

## Move Constructor with Allocator

C++

```
unordered_node_map(unordered_node_map&& other, Allocator const& a);
```

If `a == other.get_allocator()`, the element nodes of `other` are transferred directly to the new container; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

---

## Initializer List Constructor

C++

```
unordered_node_map(std::initializer_list<value_type> il,
                  size_type n = implementation-defined
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

```
unordered_node_map(size_type n, allocator_type const& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

```
unordered_node_map(size_type n, hasher const& hf, allocator_type const& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>  
unordered_node_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>  
unordered_node_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,  
                  const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate, and inserts the elements from `[f, 1)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible).

---

### initializer\_list Constructor with Allocator

```
unordered_node_map(std::initializer_list<value_type> il, const allocator_type& a);
```

C++

Constructs an empty container using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Allocator

```
unordered_node_map(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### initializer\_list Constructor with Bucket Count and Hasher and Allocator

```
unordered_node_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a);
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and default key equality predicate, and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Destructor

```
~unordered_node_map();
```

C++

**Note:** The destructor is applied to every element, and all memory is deallocated

---

### Assignment

## Copy Assignment

```
unordered_node_map& operator=(unordered_node_map const& other);
```

C++

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from `other`, copy-assigns the allocator from `other` if `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, and finally inserts copies of the elements of `other`.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

---

## Move Assignment

```
unordered_node_map& operator=(unordered_node_map&& other)
    noexcept((boost::allocator_traits<Allocator>::is_always_equal::value ||
        boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
        std::is_same<pointer, value_type*>::value);
```

C++

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to the new container; otherwise, inserts move-constructed copies of the elements of `other`.

---

## Initializer List Assignment

```
unordered_node_map& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

## Iterators

### begin

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:** O(bucket\_count())

---

### end

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

**Returns:** An iterator which refers to the past-the-end value for the container.

---

## cbegin

```
const_iterator cbegin() const noexcept;
```

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:**  $O(\text{bucket\_count}())$

---

## cend

```
const_iterator cend() const noexcept;
```

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

**Returns:** `size() == 0`

---

### size

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

**Returns:** `std::distance(begin(), end())`

---

### max\_size

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

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

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

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

### emplace\_hint

```
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
```

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

`position` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Copy Insert

```
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(const init_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(x)`, where `x` is equally convertible to both `const value_type&` and `const init_type&`, is not ambiguous and selects the `init_type` overload.

---

## Move Insert

`std::pair<iterator, bool> insert(value_type&& obj);`  
`std::pair<iterator, bool> insert(init_type&& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

A call of the form `insert(x)`, where `x` is equally convertible to both `value_type&&` and `init_type&&`, is not ambiguous and selects the `init_type` overload.

---

## Copy Insert with Hint

`iterator insert(const_iterator hint, const value_type& obj);`  
`iterator insert(const_iterator hint, const init_type& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.  
A call of the form `insert(hint, x)`, where `x` is equally convertible to both `const value_type&` and `const init_type&`, is not ambiguous and selects the `init_type` overload.

---

## Move Insert with Hint

iterator `insert(const_iterator hint, value_type&& obj);`  
iterator `insert(const_iterator hint, init_type&& obj);`

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.  
If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.  
A call of the form `insert(hint, x)`, where `x` is equally convertible to both `value_type&&` and `init_type&&`, is not ambiguous and selects the `init_type` overload.

---

## Insert Iterator Range

`template<class InputIterator> void insert(InputIterator first, InputIterator last);`

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into the container from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Initializer List

```
void insert(std::initializer_list<value_type>);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Node

```
insert_return_type insert(node_type&& nh);
```

C++

If `nh` is not empty, inserts the associated element in the container if and only if there is no element in the container with a key equivalent to `nh.key()`. `nh` is empty when the function returns.

**Returns:** An `insert_return_type` object constructed from `position`, `inserted` and `node`:

- If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty.
- Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty.
- If the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Behavior is undefined if `nh` is not empty and the allocators of `nh` and the container are not equal.

---

## Insert Node with Hint

```
iterator insert(const_iterator hint, node_type&& nh);
```

C++

If `nh` is not empty, inserts the associated element in the container if and only if there is no element in the container with a key equivalent to `nh.key()`. `nh` becomes empty if insertion took place, otherwise it is not changed.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** The iterator returned is `end()` if `nh` is empty. If insertion took place, then the iterator points to the newly inserted element; otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Behavior is undefined if `nh` is not empty and the allocators of `nh` and the container are not equal.

---

## try\_emplace

C++

```
template<class... Args>
std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
std::pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args>
std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
```

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similiar to `emplace`, with the difference that no `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

C++

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

unlike `emplace`, which simply forwards all arguments to `value_type`'s constructor.

Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## try\_emplace with Hint

C++

```
template<class... Args>
    iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
    iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class K, class... Args>
    iterator try_emplace(const_iterator hint, K&& k, Args&&... args);
```

Inserts a new element into the container if there is no existing element with key `k` contained within it.

If there is an existing element with key `k` this function does nothing.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** This function is similiar to `emplace_hint`, with the difference that no `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

unlike `emplace_hint`, which simply forwards all arguments to `value_type`'s constructor.

Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## insert\_or\_assign

C++

```
template<class M>
    std::pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
    std::pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class K, class M>
    std::pair<iterator, bool> insert_or_assign(K&& k, M&& obj);
```

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

C++

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## insert\_or\_assign with Hint

```
template<class M>
iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
template<class K, class M>
iterator insert_or_assign(const_iterator hint, K&& k, M&& obj);
```

C++

Inserts a new element into the container or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the container as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## Erase by Position

```
convertible-to-iterator erase(iterator position);
convertible-to-iterator erase(const_iterator position);
```

Erase the element pointed to by `position`.

**Returns:** An opaque object implicitly convertible to the `iterator` or `const_iterator` immediately following `position` prior to the erasure.

**Throws:** Nothing.

**Notes:** The opaque object returned must only be discarded or immediately converted to `iterator` or `const_iterator`.

## Erase by Key

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The template<class `K`> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Erase Range

```
iterator erase(const_iterator first, const_iterator last);
```

C++

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Nothing in this implementation (neither the `hasher` nor the `key_equal` objects are called).

---

## swap

```
void swap(unordered_node_map& other)  
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||  
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

C++

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## Extract by Position

```
node_type extract(const_iterator position);
```

C++

Extracts the element pointed to by `position`.

**Returns:** A `node_type` object holding the extracted element.

**Throws:** Nothing.

---

## Extract by Key

```
node_type erase(const key_type& k);
template<class K> node_type erase(K&& k);
```

Extracts the element with key equivalent to `k`, if it exists.

**Returns:** A `node_type` object holding the extracted element, or empty if no element was extracted.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## clear

```
void clear() noexcept;
```

Erases all elements in the container.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

---

## merge

```
template<class H2, class P2>
  void merge(unordered_node_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_node_map<Key, T, H2, P2, Allocator>&& source);
```

Transfers all the element nodes from `source` whose key is not already present in `*this`.

**Requires:** `this->get_allocator() == source.get_allocator()`.

**Notes:** Invalidates iterators to the elements transferred. If the resulting size of `*this` is greater than its original maximum load, invalidates all iterators associated to `*this`.

---

## Observers

### get\_allocator

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

**Returns:** The container's allocator.

---

## hash\_function

C++

```
hasher hash_function() const;
```

**Returns:** The container's hash function.

---

## key\_eq

C++

```
key_equal key_eq() const;
```

**Returns:** The container's key equality predicate

---

## Lookup

### find

C++

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
```

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `end()` if no such element exists.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## count

C++

```
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
```

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```

bool contains(const key_type& k) const;
template<class K>
    bool contains(const K& k) const;

```

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## equal\_range

```

std::pair<iterator, iterator> equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator> equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;

```

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## operator[]

```

mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
template<class K> mapped_type& operator[](K&& k);

```

**Effects:** If the container does not already contain an element with a key equivalent to `k`, inserts the value `std::pair<key_type const, mapped_type>(k, mapped_type())`.

**Returns:** A reference to `x.second` where `x` is the element already in the container, or the newly inserted element with a key equivalent to `k`.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

The template<class K> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## at

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
template<class K> mapped_type& at(const K& k);
template<class K> const mapped_type& at(const K& k) const;
```

C++

**Returns:** A reference to `x.second` where `x` is the (unique) element whose key is equivalent to `k`.

**Throws:** An exception object of type `std::out_of_range` if no such element is present.

**Notes:** The template<class K> overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

C++

**Returns:** The size of the bucket array.

## Hash Policy

### load\_factor

```
float load_factor() const noexcept;
```

C++

**Returns:** `static_cast<float>(size()) / static_cast<float>(bucket_count())`, or 0 if `bucket_count() == 0`.

### max\_load\_factor

```
float max_load_factor() const noexcept;
```

C++

**Returns:** Returns the container's maximum load factor.

---

## Set max\_load\_factor

C++

```
void max_load_factor(float z);
```

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with `boost::unordered_map`.

---

## max\_load

C++

```
size_type max_load() const noexcept;
```

**Returns:** The maximum number of elements the container can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the container's maximum load is at least `max_load_factor() * bucket_count()`. This number may decrease on erasure under high-load conditions.

---

## rehash

C++

```
void rehash(size_type n);
```

Changes if necessary the size of the bucket array so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array. If the provided Allocator uses fancy pointers, a default allocation is subsequently performed.

Invalidates iterators and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## reserve

C++

```
void reserve(size_type n);
```

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

### *iter-key-type*

```
template<class InputIterator>
using iter-key-type = std::remove_const_t<
    std::tuple_element_t<0, iter-value-type<InputIterator>>; // exposition only
```

### *iter-mapped-type*

```
template<class InputIterator>
using iter-mapped-type =
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

### *iter-to-alloc-type*

```
template<class InputIterator>
using iter-to-alloc-type = std::pair<
    std::add_const_t<std::tuple_element_t<0, iter-value-type<InputIterator>>,
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

## Equality Comparisons

### *operator==*

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

---

### operator!=

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_map<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

### Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_node_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_node_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

### erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_node_map<K, T, H, P, A>::size_type
erase_if(unordered_node_map<K, T, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
C++
```

```
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

## Serialization

`unordered_node_map` `s` can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_node_map` to an archive

Saves all the elements of an `unordered_node_map` `x` to an archive (XML archive) `ar`.

**Requires:** `std::remove_const<key_type>::type` and `std::remove_const<mapped_type>::type` are serializable (XML serializable), and they do support Boost.Serialization `save_construct_data / load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

### Loading an `unordered_node_map` from an archive

Deletes all preexisting elements of an `unordered_node_map` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_node_map` `other` saved to the storage read by `ar`.

**Requires:** `key_type` and `mapped_type` are constructible from `std::remove_const<key_type>::type&&` and `std::remove_const<mapped_type>::type&&`, respectively. `x.key_equal()` is functionally equivalent to `other.key_equal()`.

---

### Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be an `end()` iterator.

**Requires:** The `unordered_node_map` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

### Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_node_map` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template `unordered_node_set`

`boost::unordered_node_set` — A node-based, open-addressing unordered associative container that stores unique values.

`boost::unordered_node_set` uses an open-addressing layout like `boost::unordered_flat_set`, but, being node-based, it provides pointer/iterator stability and node handling functionalities. Its performance lies between those of `boost::unordered_set` and `boost::unordered_flat_set`.

As a result of its using open addressing, the interface of `boost::unordered_node_set` deviates in a number of aspects from that of `boost::unordered_set / std::unordered_set`:

- `begin()` is not constant-time.
- There is no API for bucket handling (except `bucket_count`).
- The maximum load factor of the container is managed internally and can't be set by the user.

Other than this, `boost::unordered_node_set` is mostly a drop-in replacement of standard unordered associative containers.

## Synopsis

```

// #include <boost/unordered/unordered_node_set.hpp>

namespace boost {
    template<class Key,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<Key>>
    class unordered_node_set {
public:
    // types
    using key_type           = Key;
    using value_type          = Key;
    using init_type           = Key;
    using hasher               = Hash;
    using key_equal            = Pred;
    using allocator_type       = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer         = typename std::allocator_traits<Allocator>::const_pointer;
    using reference             = value_type&;
    using const_reference       = const value_type&;
    using size_type             = std::size_t;
    using difference_type      = std::ptrdiff_t;

    using iterator              = implementation-defined;
    using const_iterator         = implementation-defined;

    using node_type             = implementation-defined;
    using insert_return_type     = implementation-defined;

    // construct/copy/destroy
    unordered_node_set();
    explicit unordered_node_set(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_node_set(const unordered_node_set& other);
    unordered_node_set(unordered_node_set&& other);
    template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l, const allocator_type& a);
    explicit unordered_node_set(const Allocator& a);
    unordered_node_set(const unordered_node_set& other, const Allocator& a);
    unordered_node_set(unordered_node_set&& other, const Allocator& a);
    unordered_node_set(std::initializer_list<value_type> il,
                      size_type n = implementation-defined
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_node_set(size_type n, const allocator_type& a);
    unordered_node_set(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
    template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
    unordered_node_set(std::initializer_list<value_type> il, const allocator_type& a);
    unordered_node_set(std::initializer_list<value_type> il, size_type n,
                      const allocator_type& a);

```

```

unordered_node_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                  const allocator_type& a);
~unordered_node_set();
unordered_node_set& operator=(const unordered_node_set& other);
unordered_node_set& operator=(unordered_node_set&& other) noexcept(
    (boost::allocator_traits<Allocator>::is_always_equal::value || 
     boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
     std::is_same<pointer, value_type*>::value);
unordered_node_set& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator      begin() noexcept;
const_iterator begin() const noexcept;
iterator      end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> std::pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
std::pair<iterator, bool> insert(const value_type& obj);
std::pair<iterator, bool> insert(value_type&& obj);
template<class K> std::pair<iterator, bool> insert(K&& k);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class K> iterator insert(const_iterator hint, K&& k);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type>);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

convertible-to-iterator      erase(iterator position);
convertible-to-iterator      erase(const_iterator position);
size_type                   erase(const key_type& k);
template<class K> size_type erase(K&& k);
iterator      erase(const_iterator first, const_iterator last);
void         swap(unordered_node_set& other)
noexcept(boost::allocator_traits<Allocator>::is_always_equal::value || 
        boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
node_type extract(const_iterator position);
node_type extract(const key_type& key);
template<class K> node_type extract(K&& key);
void         clear() noexcept;

template<class H2, class P2>
void merge(unordered_node_set<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_node_set<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;

```

```

template<class K>
    iterator      find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
size_type      count(const key_type& k) const;
template<class K>
    size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
    bool          contains(const K& k) const;
std::pair<iterator, iterator>      equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>  equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>      equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator>  equal_range(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-value-type<InputIterator>>,
         class Pred = std::equal_to<iter-value-type<InputIterator>>,
         class Allocator = std::allocator<iter-value-type<InputIterator>>>
unordered_node_set(InputIterator, InputIterator, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_node_set<iter-value-type<InputIterator>, Hash, Pred, Allocator>;

template<class T, class Hash = boost::hash<T>, class Pred = std::equal_to<T>,
         class Allocator = std::allocator<T>>
unordered_node_set(std::initializer_list<T>, typename see below::size_type = see below,
                  Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_node_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_node_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_node_set<iter-value-type<InputIterator>,
               boost::hash<iter-value-type<InputIterator>>,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_node_set(InputIterator, InputIterator, Allocator)
-> unordered_node_set<iter-value-type<InputIterator>,
               boost::hash<iter-value-type<InputIterator>>,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_node_set(InputIterator, InputIterator, typename see below::size_type, Hash,
                  Allocator)
-> unordered_node_set<iter-value-type<InputIterator>, Hash,
                  std::equal_to<iter-value-type<InputIterator>>, Allocator>;

template<class T, class Allocator>

```

```

unordered_node_set(std::initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_node_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Allocator>
unordered_node_set(std::initializer_list<T>, Allocator)
-> unordered_node_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
unordered_node_set(std::initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_node_set<T, Hash, std::equal_to<T>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_set<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_set<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
          unordered_node_set<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_node_set<K, T, H, P, A>::size_type
erase_if(unordered_node_set<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	Key must be <u>Erasable</u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container.
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <code>std::size_t</code> .
<i>Pred</i>	A binary function object that induces an equivalence relation on values of type <i>Key</i> . It takes two arguments of type <i>Key</i> and returns a value of type <code>bool</code> .
<i>Allocator</i>	An allocator whose value type is the same as the container's value type. Allocators using <u>fancy pointers</u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The element nodes of the container are held into an internal *bucket array*. A node is inserted into a bucket determined by the hash code of its element, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert` / `emplace`, or as a result of calling `rehash` / `reserve`. The *load factor* of the container (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

---

## TypeDefs

C++

```
typedef implementation-defined iterator;
```

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

Convertible to `const_iterator`.

---

C++

```
typedef implementation-defined const_iterator;
```

A constant iterator whose value type is `value_type`.

The iterator category is at least a forward iterator.

---

C++

```
typedef implementation-defined node_type;
```

A class for holding extracted container elements, modelling [NodeHandle](#) ([https://en.cppreference.com/w/cpp/container/node\\_handle](https://en.cppreference.com/w/cpp/container/node_handle)).

---

C++

```
typedef implementation-defined insert_return_type;
```

A specialization of an internal class template:

C++

```
template<class Iterator, class NodeType>
struct insert_return_type // name is exposition only
{
    Iterator position;
    bool inserted;
    NodeType node;
};
```

with `Iterator = iterator` and `NodeType = node_type`.

---

## Constructors

### Default Constructor

```
unordered_node_set();
```

C++

Constructs an empty container using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit unordered_node_set(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Iterator Range Constructor

```
template<class InputIterator>
unordered_node_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

C++

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Copy Constructor

```
unordered_node_set(unordered_node_set const& other);
```

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

## Move Constructor

```
unordered_node_set(unordered_node_set&& other);
```

The move constructor. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

## Iterator Range Constructor with Allocator

```
template<class InputIterator>
unordered_node_set(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty container using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

## Allocator Constructor

```
explicit unordered_node_set(Allocator const& a);
```

Constructs an empty container, using allocator `a`.

## Copy Constructor with Allocator

```
unordered_node_set(unordered_node_set const& other, Allocator const& a);
```

Constructs a container, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

## Move Constructor with Allocator

```
unordered_node_set(unordered_node_set&& other, Allocator const& a);
```

If `a == other.get_allocator()`, the element nodes of `other` are transferred directly to the new container; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

---

## Initializer List Constructor

C++

```
unordered_node_set(std::initializer_list<value_type> il,
                  size_type n = implementation-defined
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

C++

```
unordered_node_set(size_type n, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

C++

```
unordered_node_set(size_type n, hasher const& hf, allocator_type const& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)  
[\(https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible\).](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>
    unordered_node_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                      const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

```
unordered_node_set(std::initializer_list<value_type> il, const allocator_type& a);
```

Constructs an empty container using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)  
[\(https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible\).](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)

---

## initializer\_list Constructor with Bucket Count and Allocator

```
unordered_node_set(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)  
[\(https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible\).](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)

---

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

```
unordered_node_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                  const allocator_type& a);
```

Constructs an empty container with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and default key equality predicate, and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Destructor

```
~unordered_node_set();
```

**Note:** The destructor is applied to every element, and all memory is deallocated

---

## Assignment

### Copy Assignment

```
unordered_node_set& operator=(unordered_node_set const& other);
```

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from `other`, copy-assigns the allocator from `other` if `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, and finally inserts copies of the elements of `other`.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

---

## Move Assignment

```
unordered_node_set& operator=(unordered_node_set&& other)
noexcept((boost::allocator_traits<Allocator>::is_always_equal::value ||
            boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
            std::is_same<pointer, value_type*>::value);
```

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to the new container; otherwise, inserts move-constructed copies of the elements of `other`.

---

## Initializer List Assignment

```
unordered_node_set& operator=(std::initializer_list<value_type> il);
```

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

## Iterators

### begin

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

C++

**Returns:** An iterator referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:**  $O(\text{bucket\_count}())$

---

### end

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

C++

**Returns:** An iterator which refers to the past-the-end value for the container.

---

### cbegin

```
const_iterator cbegin() const noexcept;
```

C++

**Returns:** A `const_iterator` referring to the first element of the container, or if the container is empty the past-the-end value for the container.

**Complexity:**  $O(\text{bucket\_count}())$

---

### cend

```
const_iterator cend() const noexcept;
```

C++

**Returns:** A `const_iterator` which refers to the past-the-end value for the container.

---

## Size and Capacity

### empty

```
[[nodiscard]] bool empty() const noexcept;
```

C++

**Returns:** `size() == 0`

---

## size

`size_type size() const noexcept;`

C++

**Returns:** `std::distance(begin(), end())`

---

## max\_size

`size_type max_size() const noexcept;`

C++

**Returns:** `size()` of the largest possible container.

---

## Modifiers

### emplace

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

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

### emplace\_hint

`template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);`

C++

Inserts an object, constructed with the arguments `args`, in the container if and only if there is no element in the container with an equivalent key.

`position` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is constructible from `args`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Copy Insert

```
std::pair<iterator, bool> insert(const value_type& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Move Insert

```
std::pair<iterator, bool> insert(value_type&& obj);
```

C++

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Transparent Insert

```
template<class K> std::pair<iterator, bool> insert(K&& k);
```

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `K`.

**Returns:** The `bool` component of the return type is true if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## Copy Insert with Hint

```
iterator insert(const_iterator hint, const value_type& obj);
```

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

## Move Insert with Hint

```
iterator insert(const_iterator hint, value_type&& obj);
```

Inserts `obj` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Transparent Insert with Hint

C++

```
template<class K> std::pair<iterator, bool> insert(const_iterator hint, K&& k);
```

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.

**Returns:** The `bool` component of the return type is `true` if an insert took place.

If an insert took place, then the iterator points to the newly inserted element. Otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert Iterator Range

C++

```
template<class InputIterator> void insert(InputIterator first, InputIterator last);
```

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) into the container from `*first`.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Initializer List

```
void insert(std::initializer_list<value_type>);
```

C++

Inserts a range of elements into the container. Elements are inserted if and only if there is no element in the container with an equivalent key.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)) into the container.

**Throws:** When inserting a single element, if an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Can invalidate iterators, but only if the insert causes the load to be greater than the maximum load.

---

## Insert Node

```
insert_return_type insert(node_type&& nh);
```

C++

If `nh` is not empty, inserts the associated element in the container if and only if there is no element in the container with a key equivalent to `nh.value()`. `nh` is empty when the function returns.

**Returns:** An `insert_return_type` object constructed from `position`, `inserted` and `node`:

- If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty.
- Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty.
- If the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.value()`.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Behavior is undefined if `nh` is not empty and the allocators of `nh` and the container are not equal.

---

## Insert Node with Hint

```
iterator insert(const_iterator hint, node_type&& nh);
```

If `nh` is not empty, inserts the associated element in the container if and only if there is no element in the container with a key equivalent to `nh.value()`. `nh` becomes empty if insertion took place, otherwise it is not changed.

`hint` is a suggestion to where the element should be inserted. This implementation ignores it.

**Returns:** The iterator returned is `end()` if `nh` is empty. If insertion took place, then the iterator points to the newly inserted element; otherwise, it points to the element with equivalent key.

**Throws:** If an exception is thrown by an operation other than a call to `hasher` the function has no effect.

**Notes:** Behavior is undefined if `nh` is not empty and the allocators of `nh` and the container are not equal.

## Erase by Position

```
convertible-to-iterator erase(iterator position);
convertible-to-iterator erase(const_iterator position);
```

Erase the element pointed to by `position`.

**Returns:** An opaque object implicitly convertible to the `iterator` or `const_iterator` immediately following `position` prior to the erasure.

**Throws:** Nothing.

**Notes:** The opaque object returned must only be discarded or immediately converted to `iterator` or `const_iterator`.

## Erase by Key

```
size_type erase(const key_type& k);
template<class K> size_type erase(K&& k);
```

Erase all elements with key equivalent to `k`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## Erase Range

C++

```
iterator erase(const_iterator first, const_iterator last);
```

Erases the elements in the range from `first` to `last`.

**Returns:** The iterator following the erased elements - i.e. `last`.

**Throws:** Nothing in this implementation (neither the `hasher` nor the `key_equal` objects are called).

---

## swap

C++

```
void swap(unordered_node_set& other)  
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||  
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

Swaps the contents of the container with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## Extract by Position

C++

```
node_type extract(const_iterator position);
```

Extracts the element pointed to by `position`.

**Returns:** A `node_type` object holding the extracted element.

**Throws:** Nothing.

---

## Extract by Key

C++

```
node_type erase(const key_type& k);  
template<class K> node_type erase(K&& k);
```

Extracts the element with key equivalent to `k`, if it exists.

**Returns:** A `node_type` object holding the extracted element, or empty if no element was extracted.

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs and neither `iterator` nor `const_iterator` are implicitly convertible from `K`. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the container.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

---

## merge

```
template<class H2, class P2>
void merge(unordered_node_set<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_node_set<Key, T, H2, P2, Allocator>&& source);
```

C++

Transfers all the element nodes from `source` whose key is not already present in `*this`.

**Requires:** `this->get_allocator() == source.get_allocator()`.

**Notes:** Invalidates iterators to the elements transferred. If the resulting size of `*this` is greater than its original maximum load, invalidates all iterators associated to `*this`.

---

## Observers

### get\_allocator

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

C++

**Returns:** The container's allocator.

---

### hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The container's hash function.

---

### key\_eq

```
key_equal key_eq() const;
```

**Returns:** The container's key equality predicate

---

## Lookup

### find

```
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
template<class K>
iterator      find(const K& k);
```

**Returns:** An iterator pointing to an element with key equivalent to `k`, or `end()` if no such element exists.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## count

```
size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
```

**Returns:** The number of elements with key equivalent to `k`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## contains

```
bool      contains(const key_type& k) const;
template<class K>
bool      contains(const K& k) const;
```

**Returns:** A boolean indicating whether or not there is an element with key equal to `key` in the container

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## equal\_range

C++

```
std::pair<iterator, iterator>           equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
template<class K>
    std::pair<iterator, iterator>           equal_range(const K& k);
template<class K>
    std::pair<const_iterator, const_iterator> equal_range(const K& k) const;
```

**Returns:** A range containing all elements with key equivalent to `k`. If the container doesn't contain any such elements, returns `std::make_pair(b.end(), b.end())`.

**Notes:** The `template<class K>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bucket Interface

### bucket\_count

C++

```
size_type bucket_count() const noexcept;
```

**Returns:** The size of the bucket array.

---

## Hash Policy

### load\_factor

C++

```
float load_factor() const noexcept;
```

**Returns:** `static_cast<float>(size()) / static_cast<float>(bucket_count())`, or 0 if `bucket_count() == 0`.

---

### max\_load\_factor

C++

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the container's maximum load factor.

---

### Set max\_load\_factor

C++

```
void max_load_factor(float z);
```

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with `boost::unordered_set`.

---

## max\_load

`size_type max_load() const noexcept;`

C++

**Returns:** The maximum number of elements the container can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the container's maximum load is at least `max_load_factor() * bucket_count()`. This number may decrease on erasure under high-load conditions.

---

## rehash

`void rehash(size_type n);`

C++

Changes if necessary the size of the bucket array so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the container.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array. If the provided Allocator uses fancy pointers, a default allocation is subsequently performed.

Invalidates iterators and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

---

## reserve

`void reserve(size_type n);`

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the container.

Invalidates iterators and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the container's hash function or comparison function.

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

## Equality Comparisons

### `operator==`

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_set<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

### `operator!=`

```
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
                  const unordered_node_set<Key, T, Hash, Pred, Alloc>& y);
```

C++

Return `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Notes:** Behavior is undefined if the two containers don't have equivalent equality predicates.

## Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_node_set<Key, T, Hash, Pred, Alloc>& x,
          unordered_node_set<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));
```

C++

Swaps the contents of `x` and `y`.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the containers' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Effects:** `x.swap(y)`

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

---

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename unordered_node_set<K, T, H, P, A>::size_type
erase_if(unordered_node_set<K, T, H, P, A>& c, Predicate pred);
```

C++

Traverses the container `c` and removes all elements for which the supplied predicate returns `true`.

**Returns:** The number of erased elements.

**Notes:** Equivalent to:

```
auto original_size = c.size();
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
return original_size - c.size();
```

C++

## Serialization

`unordered_node_set`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `unordered_node_set` to an archive

Saves all the elements of an `unordered_node_set` `x` to an archive (XML archive) `ar`.

**Requires:** `value_type` is serializable (XML serializable), and it supports Boost.Serialization `save_construct_data` / `load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

---

### Loading an `unordered_node_set` from an archive

Deletes all preexisting elements of an `unordered_node_set` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `unordered_node_set` other saved to the storage read by `ar`.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)). `x.key_equal()` is functionally equivalent to `other.key_equal()`.

---

## Saving an iterator/const\_iterator to an archive

Saves the positional information of an `iterator` (`const_iterator`) `it` to an archive (XML archive) `ar`. `it` can be and `end()` iterator.

**Requires:** The `unordered_node_set` `x` pointed to by `it` has been previously saved to `ar`, and no modifying operations have been issued on `x` between saving of `x` and saving of `it`.

---

## Loading an iterator/const\_iterator from an archive

Makes an `iterator` (`const_iterator`) `it` point to the restored position of the original `iterator` (`const_iterator`) saved to the storage read by an archive (XML archive) `ar`.

**Requires:** If `x` is the `unordered_node_set` `it` points to, no modifying operations have been issued on `x` between loading of `x` and loading of `it`.

## Class Template concurrent\_flat\_map

`boost::concurrent_flat_map` — A hash table that associates unique keys with another value and allows for concurrent element insertion, erasure, lookup and access without external synchronization mechanisms.

Even though it acts as a container, `boost::concurrent_flat_map` does not model the standard C++ [Container](https://en.cppreference.com/w/cpp/named_req/Container) ([https://en.cppreference.com/w/cpp/named\\_req/Container](https://en.cppreference.com/w/cpp/named_req/Container)) concept. In particular, iterators and associated operations (`begin`, `end`, etc.) are not provided. Element access and modification are done through user-provided *visitation functions* that are passed to `concurrent_flat_map` operations where they are executed internally in a controlled fashion. Such visitation-based API allows for low-contention concurrent usage scenarios.

The internal data structure of `boost::concurrent_flat_map` is similar to that of `boost::unordered_flat_map`. As a result of its using open-addressing techniques, `value_type` must be move-constructible and pointer stability is not kept under rehashing.

## Synopsis

```

// #include <boost/unordered/concurrent_flat_map.hpp>

namespace boost {
    template<class Key,
              class T,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<std::pair<const Key, T>>>
    class concurrent_flat_map {
public:
    // types
    using key_type           = Key;
    using mapped_type         = T;
    using value_type          = std::pair<const Key, T>;
    using init_type           = std::pair<
                                typename std::remove_const<Key>::type,
                                typename std::remove_const<T>::type
                                >;
    using hasher              = Hash;
    using key_equal           = Pred;
    using allocator_type       = Allocator;
    using pointer              = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer        = typename std::allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = std::size_t;
    using difference_type      = std::ptrdiff_t;

    // constants
    static constexpr size_type bulk_visit_size = implementation-defined;

    // construct/copy/destroy
    concurrent_flat_map();
    explicit concurrent_flat_map(size_type n,
                                  const hasher& hf = hasher(),
                                  const key_equal& eql = key_equal(),
                                  const allocator_type& a = allocator_type());
    template<class InputIterator>
    concurrent_flat_map(InputIterator f, InputIterator l,
                        size_type n = implementation-defined,
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());
    concurrent_flat_map(const concurrent_flat_map& other);
    concurrent_flat_map(concurrent_flat_map&& other);
    template<class InputIterator>
    concurrent_flat_map(InputIterator f, InputIterator l, const allocator_type& a);
    explicit concurrent_flat_map(const Allocator& a);
    concurrent_flat_map(const concurrent_flat_map& other, const Allocator& a);
    concurrent_flat_map(concurrent_flat_map&& other, const Allocator& a);
    concurrent_flat_map(unordered_flat_map<Key, T, Hash, Pred, Allocator>&& other);
    concurrent_flat_map(std::initializer_list<value_type> il,
                        size_type n = implementation-defined
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());
    concurrent_flat_map(size_type n, const allocator_type& a);
    concurrent_flat_map(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    concurrent_flat_map(InputIterator f, InputIterator l, size_type n,
                        const allocator_type& a);
    template<class InputIterator>
    concurrent_flat_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,

```

```

        const allocator_type& a);
concurrent_flat_map(std::initializer_list<value_type> il, const allocator_type& a);
concurrent_flat_map(std::initializer_list<value_type> il, size_type n,
                    const allocator_type& a);
concurrent_flat_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                    const allocator_type& a);
~concurrent_flat_map();
concurrent_flat_map& operator=(const concurrent_flat_map& other);
concurrent_flat_map& operator=(concurrent_flat_map&& other) noexcept(
    (boost::allocator_traits<Allocator>::is_always_equal::value ||
     boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
     std::is_same<pointer, value_type*>::value);
concurrent_flat_map& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// visitation
template<class F> size_t visit(const key_type& k, F f);
template<class F> size_t visit(const key_type& k, F f) const;
template<class F> size_t cvisit(const key_type& k, F f) const;
template<class K, class F> size_t visit(const K& k, F f);
template<class K, class F> size_t visit(const K& k, F f) const;
template<class K, class F> size_t cvisit(const K& k, F f) const;

template<class FwdIterator, class F>
size_t visit(FwdIterator first, FwdIterator last, F f);
template<class FwdIterator, class F>
size_t visit(FwdIterator first, FwdIterator last, F f) const;
template<class FwdIterator, class F>
size_t cvisit(FwdIterator first, FwdIterator last, F f) const;

template<class F> size_t visit_all(F f);
template<class F> size_t visit_all(F f) const;
template<class F> size_t cvisit_all(F f) const;
template<class ExecutionPolicy, class F>
void visit_all(ExecutionPolicy&& policy, F f);
template<class ExecutionPolicy, class F>
void visit_all(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F>
void cvisit_all(ExecutionPolicy&& policy, F f) const;

template<class F> bool visit_while(F f);
template<class F> bool visit_while(F f) const;
template<class F> bool cvisit_while(F f) const;
template<class ExecutionPolicy, class F>
bool visit_while(ExecutionPolicy&& policy, F f);
template<class ExecutionPolicy, class F>
bool visit_while(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F>
bool cvisit_while(ExecutionPolicy&& policy, F f) const;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> bool emplace(Args&&... args);
bool insert(const value_type& obj);
bool insert(const init_type& obj);
bool insert(value_type&& obj);
bool insert(init_type&& obj);
template<class InputIterator> size_type insert(InputIterator first, InputIterator last);

```

```

size_type insert(std::initializer_list<value_type> il);

template<class... Args, class F> bool emplace_or_visit(Args&&... args, F&& f);
template<class... Args, class F> bool emplace_or_cvvisit(Args&&... args, F&& f);
template<class F> bool insert_or_visit(const value_type& obj, F f);
template<class F> bool insert_or_cvvisit(const value_type& obj, F f);
template<class F> bool insert_or_visit(const init_type& obj, F f);
template<class F> bool insert_or_cvvisit(const init_type& obj, F f);
template<class F> bool insert_or_visit(value_type&& obj, F f);
template<class F> bool insert_or_cvvisit(value_type&& obj, F f);
template<class F> bool insert_or_visit(init_type&& obj, F f);
template<class F> bool insert_or_cvvisit(init_type&& obj, F f);
template<class InputIterator,class F>
    size_type insert_or_visit(InputIterator first, InputIterator last, F f);
template<class InputIterator,class F>
    size_type insert_or_cvvisit(InputIterator first, InputIterator last, F f);
template<class F> size_type insert_or_visit(std::initializer_list<value_type> il, F f);
template<class F> size_type insert_or_cvvisit(std::initializer_list<value_type> il, F f);

template<class... Args> bool try_emplace(const key_type& k, Args&&... args);
template<class... Args> bool try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args> bool try_emplace(K&& k, Args&&... args);

template<class... Args, class F>
    bool try_emplace_or_visit(const key_type& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_cvvisit(const key_type& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_visit(key_type&& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_cvvisit(key_type&& k, Args&&... args, F&& f);
template<class K, class... Args, class F>
    bool try_emplace_or_visit(K&& k, Args&&... args, F&& f);
template<class K, class... Args, class F>
    bool try_emplace_or_cvvisit(K&& k, Args&&... args, F&& f);

template<class M> bool insert_or_assign(const key_type& k, M&& obj);
template<class M> bool insert_or_assign(key_type&& k, M&& obj);
template<class K, class M> bool insert_or_assign(K&& k, M&& obj);

size_type erase(const key_type& k);
template<class K> size_type erase(const K& k);

template<class F> size_type erase_if(const key_type& k, F f);
template<class K, class F> size_type erase_if(const K& k, F f);
template<class F> size_type erase_if(F f);
template<class ExecutionPolicy, class F> void erase_if(ExecutionPolicy&& policy, F f);

void swap(concurrent_flat_map& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
void clear() noexcept;

template<class H2, class P2>
    size_type merge(concurrent_flat_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
    size_type merge(concurrent_flat_map<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations

```

```

size_type      count(const key_type& k) const;
template<class K>
size_type      count(const K& k) const;
bool          contains(const key_type& k) const;
template<class K>
bool          contains(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-key-type<InputIterator>>,
         class Pred = std::equal_to<iter-key-type<InputIterator>>,
         class Allocator = std::allocator<iter-to-alloc-type<InputIterator>>>
concurrent_flat_map(InputIterator, InputIterator, typename see below::size_type = see below,
                     Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> concurrent_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                     Pred, Allocator>;

template<class Key, class T, class Hash = boost::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
concurrent_flat_map(std::initializer_list<std::pair<Key, T>>,
                     typename see below::size_type = see below, Hash = Hash(),
                     Pred = Pred(), Allocator = Allocator())
-> concurrent_flat_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
concurrent_flat_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> concurrent_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                     boost::hash<iter-key-type<InputIterator>>,
                     std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
concurrent_flat_map(InputIterator, InputIterator, Allocator)
-> concurrent_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                     boost::hash<iter-key-type<InputIterator>>,
                     std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
concurrent_flat_map(InputIterator, InputIterator, typename see below::size_type, Hash,
                     Allocator)
-> concurrent_flat_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                     std::equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
concurrent_flat_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                     Allocator)
-> concurrent_flat_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
concurrent_flat_map(std::initializer_list<std::pair<Key, T>>, Allocator)
-> concurrent_flat_map<Key, T, boost::hash<Key>, std::equal_to<Key>, Allocator>;

```

```

template<class Key, class T, class Hash, class Allocator>
concurrent_flat_map(std::initializer_list<std::pair<Key, T>>, typename see below::size_type,
                    Hash, Allocator)
-> concurrent_flat_map<Key, T, Hash, std::equal_to<Key>, Allocator>;

// Equality Comparisons
template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator==(const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y);

template<class Key, class T, class Hash, class Pred, class Alloc>
bool operator!=(const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y);

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
          concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class T, class H, class P, class A, class Predicate>
typename concurrent_flat_map<K, T, H, P, A>::size_type
erase_if(concurrent_flat_map<K, T, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	<i>Key</i> and <i>T</i> must be <u><a href="#">MoveConstructible</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/MoveConstructible">https://en.cppreference.com/w/cpp/named_req/MoveConstructible</a> ). <i>std::pair&lt;const Key, T&gt;</i> must be <u><a href="#">EmplaceConstructible</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible">https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible</a> ) into the table from any <i>std::pair</i> object convertible to it, and it also must be <u><a href="#">Erasable</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the table.
<i>T</i>	
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <i>std::size_t</i> .
<i>Pred</i>	A binary function object that induces an equivalence relation on values of type <i>Key</i> . It takes two arguments of type <i>Key</i> and returns a value of type <i>bool</i> .
<i>Allocator</i>	An allocator whose value type is the same as the table's value type. Allocators using <u><a href="#">fancy pointers</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers">https://en.cppreference.com/w/cpp/named_req/Allocator#Fancy_pointers</a> ) are supported.

The elements of the table are held into an internal *bucket array*. An element is inserted into a bucket determined by its hash code, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert / emplace`, or as a result of calling `rehash / reserve`. The *load factor* of the table (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

---

## Concurrency Requirements and Guarantees

Concurrent invocations of `operator()` on the same `const` instance of `Hash` or `Pred` are required to not introduce data races. For `Alloc` being either `Allocator` or any allocator type rebound from `Allocator`, concurrent invocations of the following operations on the same instance `al` of `Alloc` are required to not introduce data races:

- Copy construction from `al` of an allocator rebound from `Alloc`
- `std::allocator_traits<Alloc>::allocate`
- `std::allocator_traits<Alloc>::deallocate`
- `std::allocator_traits<Alloc>::construct`
- `std::allocator_traits<Alloc>::destroy`

In general, these requirements on `Hash`, `Pred` and `Allocator` are met if these types are not stateful or if the operations only involve constant access to internal data members.

With the exception of destruction, concurrent invocations of any operation on the same instance of a `concurrent_flat_map` do not introduce data races — that is, they are thread-safe.

If an operation `op` is explicitly designated as *blocking on* `x`, where `x` is an instance of a `boost::concurrent_flat_map`, prior blocking operations on `x` synchronize with `op`. So, blocking operations on the same `concurrent_flat_map` execute sequentially in a multithreaded scenario.

An operation is said to be *blocking on rehashing of* `x` if it blocks on `x` only when an internal rehashing is issued.

Access or modification of an element of a `boost::concurrent_flat_map` passed by reference to a user-provided visitation function do not introduce data races when the visitation function is executed internally by the `boost::concurrent_flat_map`.

Any `boost::concurrent_flat_map` operation that inserts or modifies an element `e` synchronizes with the internal invocation of a visitation function on `e`.

Visitation functions executed by a `boost::concurrent_flat_map` `x` are not allowed to invoke any operation on `x`; invoking operations on a different `boost::concurrent_flat_map` instance `y` is allowed only if concurrent outstanding operations on `y` do not access `x` directly or indirectly.

---

## Configuration Macros

## BOOST\_UNORDERED\_DISABLE\_REENTRANCY\_CHECK

In debug builds (more precisely, when `BOOST_ASSERT_IS_VOID` is not defined), *container reentrancies* (illegally invoking an operation on `m` from within a function visiting elements of `m`) are detected and signalled through `BOOST_ASSERT_MSG`. When run-time speed is a concern, the feature can be disabled by globally defining this macro.

## Constants

```
static constexpr size_type bulk_visit_size;
```

CPP

Chunk size internally used in bulk visit operations.

## Constructors

### Default Constructor

```
concurrent_flat_map();
```

C++

Constructs an empty table using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

```
explicit concurrent_flat_map(size_type n,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a = allocator_type());
```

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Iterator Range Constructor

```
template<class InputIterator>
concurrent_flat_map(InputIterator f, InputIterator l,
                    size_type n = implementation-defined,
                    const hasher& hf = hasher(),
                    const key_equal& eql = key_equal(),
                    const allocator_type& a = allocator_type());
```

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

C++

```
concurrent_flat_map(concurrent_flat_map const& other);
```

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

**Concurrency:** Blocking on `other`.

---

## Move Constructor

C++

```
concurrent_flat_map(concurrent_flat_map&& other);
```

The move constructor. The internal bucket array of `other` is transferred directly to the new table. The hash function, predicate and allocator are moved-constructed from `other`.

**Concurrency:** Blocking on `other`.

---

## Iterator Range Constructor with Allocator

C++

```
template<class InputIterator>
concurrent_flat_map(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty table using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

C++

```
explicit concurrent_flat_map(Allocator const& a);
```

Constructs an empty table, using allocator `a`.

---

## Copy Constructor with Allocator

```
concurrent_flat_map(concurrent_flat_map const& other, Allocator const& a);
```

C++

Constructs a table, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

**Concurrency:** Blocking on `other`.

---

## Move Constructor with Allocator

```
concurrent_flat_map(concurrent_flat_map&& other, Allocator const& a);
```

C++

If `a == other.get_allocator()`, the elements of `other` are transferred directly to the new table; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

**Concurrency:** Blocking on `other`.

---

## Move Constructor from unordered\_flat\_map

```
concurrent_flat_map(unordered_flat_map<Key, T, Hash, Pred, Allocator>&& other);
```

C++

Move construction from a `unordered_flat_map`. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

**Complexity:**  $O(\text{bucket\_count}())$

---

## Initializer List Constructor

```
concurrent_flat_map(std::initializer_list<value_type> il,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be DefaultConstructible ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

```
concurrent_flat_map(size_type n, allocator_type const& a);
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

```
concurrent_flat_map(size_type n, hasher const& hf, allocator_type const& a);
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

```
template<class InputIterator>
concurrent_flat_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty table with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

```
template<class InputIterator>
concurrent_flat_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                    const allocator_type& a);
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `a` as the allocator, with the default key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

```
concurrent_flat_map(std::initializer_list<value_type> il, const allocator_type& a);
```

C++

Constructs an empty table using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Allocator

```
concurrent_flat_map(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

C++

Constructs an empty table with at least `n` buckets, using `a` and default hash function and key equality predicate, and inserts the elements from `il` into it.

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#)

([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

```
concurrent_flat_map(std::initializer_list<value_type> il, size_type n, const hasher& hf,  
                    const allocator_type& a);
```

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `a` as the allocator and default key equality predicate, and inserts the elements from `il` into it.

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Destructor

```
~concurrent_flat_map();
```

C++

**Note:** The destructor is applied to every element, and all memory is deallocated

---

## Assignment

### Copy Assignment

```
concurrent_flat_map& operator=(concurrent_flat_map const& other);
```

C++

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from `other`, copy-assigns the allocator from `other` if `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, and finally inserts copies of the elements of `other`.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

**Concurrency:** Blocking on `*this` and `other`.

---

## Move Assignment

```
concurrent_flat_map& operator=(concurrent_flat_map&& other)
    noexcept((boost::allocator_traits<Allocator>::is_always_equal::value ||
        boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value) &&
        std::is_same<pointer, value_type*>::value);
```

C++

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to `*this`; otherwise, inserts move-constructed copies of the elements of `other`.

**Concurrency:** Blocking on `*this` and `other`.

---

## Initializer List Assignment

```
concurrent_flat_map& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

**Concurrency:** Blocking on `*this`.

---

## Visitation

### [c]visit

```
template<class F> size_t visit(const key_type& k, F f);
template<class F> size_t visit(const key_type& k, F f) const;
template<class F> size_t cvisit(const key_type& k, F f) const;
template<class K, class F> size_t visit(const K& k, F f);
template<class K, class F> size_t visit(const K& k, F f) const;
template<class K, class F> size_t cvisit(const K& k, F f) const;
```

C++

If an element `x` exists with key equivalent to `k`, invokes `f` with a reference to `x`. Such reference is `const` iff `*this` is `const`.

**Returns:** The number of elements visited (0 or 1).

**Notes:** The template<class K, class F> overloads only participate in overload resolution if Hash::is\_transparent and Pred::is\_transparent are valid member typedefs. The library assumes that Hash is callable with both K and Key and that Pred is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the Key type.

---

## Bulk visit

```
template<class FwdIterator, class F>
size_t visit(FwdIterator first, FwdIterator last, F f);
template<class FwdIterator, class F>
size_t visit(FwdIterator first, FwdIterator last, F f) const;
template<class FwdIterator, class F>
size_t cvisit(FwdIterator first, FwdIterator last, F f) const;
```

C++

For each element k in the range [first, last), if there is an element x in the container with key equivalent to k , invokes f with a reference to x . Such reference is const iff \*this is const.

Although functionally equivalent to individually invoking [c]visit for each key, bulk visitation performs generally faster due to internal streamlining optimizations. It is advisable that std::distance(first, last) be at least bulk\_visit\_size to enjoy a performance gain: beyond this size, performance is not expected to increase further.

**Requires:** FwdIterator is a [LegacyForwardIterator](https://en.cppreference.com/w/cpp/named_req/ForwardIterator) (https://en.cppreference.com/w/cpp/named\_req/ForwardIterator) (C++11 to C++17), or satisfies [std::forward\\_iterator](https://en.cppreference.com/w/cpp/iterator/forward_iterator) (https://en.cppreference.com/w/cpp/iterator/forward\_iterator) (C++20 and later). For K = std::iterator\_traits<FwdIterator>::value\_type , either K is key\_type or else Hash::is\_transparent and Pred::is\_transparent are valid member typedefs. In the latter case, the library assumes that Hash is callable with both K and Key and that Pred is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the Key type.

**Returns:** The number of elements visited.

---

## [c]visit\_all

```
template<class F> size_t visit_all(F f);
template<class F> size_t visit_all(F f) const;
template<class F> size_t cvisit_all(F f) const;
```

C++

Successively invokes f with references to each of the elements in the table. Such references are const iff \*this is const.

**Returns:** The number of elements visited.

---

## Parallel [c]visit\_all

```
template<class ExecutionPolicy, class F> void visit_all(ExecutionPolicy&& policy, F f);
template<class ExecutionPolicy, class F> void visit_all(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F> void cvisit_all(ExecutionPolicy&& policy, F f) const;
```

C++

Invokes `f` with references to each of the elements in the table. Such references are const iff `*this` is const. Execution is parallelized according to the semantics of the execution policy specified.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

These overloads only participate in overload resolution if  
`std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is true.

Unsequenced execution policies are not allowed.

---

### [c]visit\_while

```
template<class F> bool visit_while(F f);
template<class F> bool visit_while(F f) const;
template<class F> bool cvisit_while(F f) const;
```

C++

Successively invokes `f` with references to each of the elements in the table until `f` returns `false` or all the elements are visited. Such references to the elements are const iff `*this` is const.

**Returns:** `false` iff `f` ever returns `false`.

---

### Parallel [c]visit\_while

```
template<class ExecutionPolicy, class F> bool visit_while(ExecutionPolicy&& policy, F f);
template<class ExecutionPolicy, class F> bool visit_while(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F> bool cvisit_while(ExecutionPolicy&& policy, F f) const;
```

C++

Invokes `f` with references to each of the elements in the table until `f` returns `false` or all the elements are visited. Such references to the elements are const iff `*this` is const. Execution is parallelized according to the semantics of the execution policy specified.

**Returns:** `false` iff `f` ever returns `false`.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

These overloads only participate in overload resolution if  
`std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true`.

Unsequenced execution policies are not allowed.

Parallelization implies that execution does not necessarily finish as soon as `f` returns `false`, and as a result `f` may be invoked with further elements for which the return value is also `false`.

---

## Size and Capacity

### empty

C++  
[[nodiscard]] `bool empty() const noexcept;`

**Returns:** `size() == 0`

---

### size

C++  
`size_type size() const noexcept;`

**Returns:** The number of elements in the table.

**Notes:** In the presence of concurrent insertion operations, the value returned may not accurately reflect the true size of the table right after execution.

---

### max\_size

C++  
`size_type max_size() const noexcept;`

**Returns:** `size()` of the largest possible table.

---

## Modifiers

### emplace

C++  
`template<class... Args> bool emplace(Args&&... args);`

Inserts an object, constructed with the arguments `args`, in the table if and only if there is no element in the table with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Copy Insert

bool **insert**(const value\_type& obj);  
bool **insert**(const init\_type& obj);

C++

Inserts obj in the table if and only if there is no element in the table with an equivalent key.

**Requires:** value\_type is CopyInsertable ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

A call of the form `insert(x)`, where x is equally convertible to both `const value_type&` and `const init_type&`, is not ambiguous and selects the `init_type` overload.

---

## Move Insert

bool **insert**(value\_type&& obj);  
bool **insert**(init\_type&& obj);

C++

Inserts obj in the table if and only if there is no element in the table with an equivalent key.

**Requires:** value\_type is MoveInsertable ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

A call of the form `insert(x)`, where x is equally convertible to both `value_type&&` and `init_type&&`, is not ambiguous and selects the `init_type` overload.

---

## Insert Iterator Range

template<class InputIterator> size\_type **insert**(InputIterator first, InputIterator last);

C++

Equivalent to

```
while(first != last) this->emplace(*first++);
```

**Returns:** The number of elements inserted.

---

## Insert Initializer List

```
size_type insert(std::initializer_list<value_type> il);
```

C++

Equivalent to

```
this->insert(il.begin(), il.end());
```

**Returns:** The number of elements inserted.

---

## emplace\_or\_[c]visit

```
template<class... Args, class F> bool emplace_or_visit(Args&&... args, F&& f);
template<class... Args, class F> bool emplace_or_cvisit(Args&&... args, F&& f);
```

C++

Inserts an object, constructed with the arguments `args`, in the table if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a reference to the equivalent element; such reference is const iff `emplace_or_cvisit` is used.

**Requires:** `value_type` is constructible from `args`.

**Returns:** `true` if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

The interface is exposition only, as C++ does not allow to declare a parameter `f` after a variadic parameter pack.

---

## Copy insert\_or\_[c]visit

```
template<class F> bool insert_or_visit(const value_type& obj, F f);
template<class F> bool insert_or_cvisit(const value_type& obj, F f);
template<class F> bool insert_or_visit(const init_type& obj, F f);
template<class F> bool insert_or_cvisit(const init_type& obj, F f);
```

C++

Inserts `obj` in the table if and only if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a reference to the equivalent element; such reference is const iff a `*_cvisit` overload is used.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** `true` if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

In a call of the form `insert_or_[c]visit(obj, f)`, the overloads accepting a `const value_type&` argument participate in overload resolution only if  
`std::remove_cv<std::remove_reference<decltype(obj)>::type>::type` is `value_type`.

---

## Move insert\_or\_[c]visit

C++

```
template<class F> bool insert_or_visit(value_type&& obj, F f);
template<class F> bool insert_or_cvisit(value_type&& obj, F f);
template<class F> bool insert_or_visit(init_type&& obj, F f);
template<class F> bool insert_or_cvisit(init_type&& obj, F f);
```

Inserts `obj` in the table if and only if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a reference to the equivalent element; such reference is `const` iff a `*_cvisit` overload is used.

**Requires:** `value_type` is [MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** `true` if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

In a call of the form `insert_or_[c]visit(obj, f)`, the overloads accepting a `value_type&&` argument participate in overload resolution only if `std::remove_reference<decltype(obj)>::type` is `value_type`.

---

## Insert Iterator Range or Visit

C++

```
template<class InputIterator, class F>
size_type insert_or_visit(InputIterator first, InputIterator last, F f);
template<class InputIterator, class F>
size_type insert_or_cvisit(InputIterator first, InputIterator last, F f);
```

Equivalent to

```
while(first != last) this->emplace_or_[c]visit(*first++, f);
```

**Returns:** The number of elements inserted.

## Insert Initializer List or Visit

```
template<class F> size_type insert_or_visit(std::initializer_list<value_type> il, F f);
template<class F> size_type insert_or_cvisit(std::initializer_list<value_type> il, F f);
```

C++

Equivalent to

```
this->insert_or[c]visit(il.begin(), il.end(), f);
```

**Returns:** The number of elements inserted.

---

## try\_emplace

```
template<class... Args> bool try_emplace(const key_type& k, Args&&... args);
template<class... Args> bool try_emplace(key_type&& k, Args&&... args);
template<class K, class... Args> bool try_emplace(K&& k, Args&&... args);
```

C++

Inserts an element constructed from `k` and `args` into the table if there is no existing element with key `k` contained within it.

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** This function is similiar to `emplace`, with the difference that no `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))
```

C++

unlike `emplace`, which simply forwards all arguments to `value_type`'s constructor.

Invalidates pointers and references to elements if a rehashing is issued.

The `template<class K, class... Args>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## try\_emplace\_or\_[c]visit

```

template<class... Args, class F>
    bool try_emplace_or_visit(const key_type& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_cvvisit(const key_type& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_visit(key_type&& k, Args&&... args, F&& f);
template<class... Args, class F>
    bool try_emplace_or_cvvisit(key_type&& k, Args&&... args, F&& f);
template<class K, class... Args, class F>
    bool try_emplace_or_visit(K&& k, Args&&... args, F&& f);
template<class K, class... Args, class F>
    bool try_emplace_or_cvvisit(K&& k, Args&&... args, F&& f);

```

Inserts an element constructed from `k` and `args` into the table if there is no existing element with key `k` contained within it. Otherwise, invokes `f` with a reference to the equivalent element; such reference is const iff a `*_cvvisit` overload is used.

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** No `value_type` is constructed if there is an element with an equivalent key; otherwise, the construction is of the form:

```

// first four overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

// last two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<Args>(args)...))

```

Invalidate pointers and references to elements if a rehashing is issued.

The interface is exposition only, as C++ does not allow to declare a parameter `f` after a variadic parameter pack.

The `template<class K, class... Args, class F>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

## insert\_or\_assign

```

template<class M> bool insert_or_assign(const key_type& k, M&& obj);
template<class M> bool insert_or_assign(key_type&& k, M&& obj);
template<class K, class M> bool insert_or_assign(K&& k, M&& obj);

```

Inserts a new element into the table or updates an existing one by assigning to the contained value.

If there is an element with key `k`, then it is updated by assigning `std::forward<M>(obj)`.

If there is no such element, it is added to the table as:

```
// first two overloads
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<Key>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))

// third overload
value_type(std::piecewise_construct,
           std::forward_as_tuple(std::forward<K>(k)),
           std::forward_as_tuple(std::forward<M>(obj)))
```

C++

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

The `template<class K, class M>` only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## erase

```
size_type erase(const key_type& k);
template<class K> size_type erase(const K& k);
```

C++

Erases the element with key equivalent to `k` if it exists.

**Returns:** The number of elements erased (0 or 1).

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## erase\_if by Key

```
template<class F> size_type erase_if(const key_type& k, F f);
template<class K, class F> size_type erase_if(const K& k, F f);
```

C++

Erases the element `x` with key equivalent to `k` if it exists and `f(x)` is true.

**Returns:** The number of elements erased (0 or 1).

**Throws:** Only throws an exception if it is thrown by `hasher`, `key_equal` or `f`.

**Notes:** The template<class `K`, class `F`> overload only participates in overload resolution if `std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `false`.

The template<class `K`, class `F`> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## erase\_if

```
template<class F> size_type erase_if(F f);
```

C++

Successively invokes `f` with references to each of the elements in the table, and erases those for which `f` returns `true`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `f`.

---

## Parallel erase\_if

```
template<class ExecutionPolicy, class F> void erase_if(ExecutionPolicy&& policy, F f);
```

C++

Invokes `f` with references to each of the elements in the table, and erases those for which `f` returns `true`. Execution is parallelized according to the semantics of the execution policy specified.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

This overload only participates in overload resolution if `std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true`.

Unsequenced execution policies are not allowed.

---

## swap

```
void swap(concurrent_flat_map& other)
  noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
           boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

C++

Swaps the contents of the table with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is true then the tables' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

**Concurrency:** Blocking on `*this` and `other`.

---

## clear

`void clear() noexcept;`

C++

Erases all elements in the table.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

**Concurrency:** Blocking on `*this`.

---

## merge

`template<class H2, class P2>`  
    `size_type merge(concurrent_flat_map<Key, T, H2, P2, Allocator>& source);`  
`template<class H2, class P2>`  
    `size_type merge(concurrent_flat_map<Key, T, H2, P2, Allocator>&& source);`

C++

Move-inserts all the elements from `source` whose key is not already present in `*this`, and erases them from `source`.

**Returns:** The number of elements inserted.

**Concurrency:** Blocking on `*this` and `source`.

---

## Observers

### get\_allocator

`allocator_type get_allocator() const noexcept;`

C++

**Returns:** The table's allocator.

---

### hash\_function

`hasher hash_function() const;`

C++

**Returns:** The table's hash function.

## key\_eq

C++

```
key_equal key_eq() const;
```

**Returns:** The table's key equality predicate.

---

## Map Operations

### count

C++

```
size_type count(const key_type& k) const;  
template<class K>  
size_type count(const K& k) const;
```

**Returns:** The number of elements with key equivalent to `k` (0 or 1).

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

### contains

C++

```
bool contains(const key_type& k) const;  
template<class K>  
bool contains(const K& k) const;
```

**Returns:** A boolean indicating whether or not there is an element with key equal to `k` in the table.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

## Bucket Interface

### bucket\_count

C++

```
size_type bucket_count() const noexcept;
```

**Returns:** The size of the bucket array.

---

## Hash Policy

### load\_factor

```
float load_factor() const noexcept;
```

C++

**Returns:** static\_cast<float>(size()) / static\_cast<float>(bucket\_count()), or 0 if bucket\_count() == 0.

---

### max\_load\_factor

```
float max_load_factor() const noexcept;
```

C++

**Returns:** Returns the table's maximum load factor.

---

### Set max\_load\_factor

```
void max_load_factor(float z);
```

C++

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with boost::unordered\_map .

---

### max\_load

```
size_type max_load() const noexcept;
```

C++

**Returns:** The maximum number of elements the table can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the table's maximum load is at least max\_load\_factor() \* bucket\_count() . This number may decrease on erasure under high-load conditions.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

### rehash

```
void rehash(size_type n);
```

C++

Changes if necessary the size of the bucket array so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the table.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates pointers and references to elements, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the table's hash function or comparison function.

**Concurrency:** Blocking on `*this`.

---

## reserve

`void reserve(size_type n);`

C++

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the table.

Invalidates pointers and references to elements, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the table's hash function or comparison function.

**Concurrency:** Blocking on `*this`.

---

## Deduction Guides

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the table type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

## *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

### *iter-key-type*

```
template<class InputIterator>
using iter-key-type = std::remove_const_t<
    std::tuple_element_t<0, iter-value-type<InputIterator>>>; // exposition only
```

### *iter-mapped-type*

```
template<class InputIterator>
using iter-mapped-type =
    std::tuple_element_t<1, iter-value-type<InputIterator>>; // exposition only
```

### *iter-to-alloc-type*

```
template<class InputIterator>
using iter-to-alloc-type = std::pair<
    std::add_const_t<std::tuple_element_t<0, iter-value-type<InputIterator>>>,
    std::tuple_element_t<1, iter-value-type<InputIterator>>>; // exposition only
```

## Equality Comparisons

### operator==

```
template<class Key, class T, class Hash, class Pred, class Alloc> C++
bool operator==(const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y);
```

Returns `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Concurrency:** Blocking on `x` and `y`.

**Notes:** Behavior is undefined if the two tables don't have equivalent equality predicates.

### operator!=

```
template<class Key, class T, class Hash, class Pred, class Alloc> C++
bool operator!=(const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
                  const concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y);
```

Returns `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Concurrency:** Blocking on `x` and `y`.

**Notes:** Behavior is undefined if the two tables don't have equivalent equality predicates.

## Swap

```
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(concurrent_flat_map<Key, T, Hash, Pred, Alloc>& x,
          concurrent_flat_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y))):
```

Equivalent to

```
x.swap(y);
```

## erase\_if

```
template<class K, class T, class H, class P, class A, class Predicate>
typename concurrent_flat_map<K, T, H, P, A>::size_type
erase_if(concurrent_flat_map<K, T, H, P, A>& c, Predicate pred);
```

Equivalent to

```
c.erase_if(pred);
```

## Serialization

`concurrent_flat_map`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `concurrent_flat_map` to an archive

Saves all the elements of a `concurrent_flat_map` `x` to an archive (XML archive) `ar`.

**Requires:** `std::remove_const<key_type>::type` and `std::remove_const<mapped_type>::type` are serializable (XML serializable), and they do support Boost.Serialization `save_construct_data` / `load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

**Concurrency:** Blocking on `x`.

### Loading an `concurrent_flat_map` from an archive

Deletes all preexisting elements of a `concurrent_flat_map` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `concurrent_flat_map` `other` saved to the storage read by `ar`.

**Requires:** `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Concurrency:** Blocking on `x`.

## Class Template `concurrent_flat_set`

`boost::concurrent_flat_set` — A hash table that stores unique values and allows for concurrent element insertion, erasure, lookup and access without external synchronization mechanisms.

Even though it acts as a container, `boost::concurrent_flat_set` does not model the standard C++ [Container](#) ([https://en.cppreference.com/w/cpp/named\\_req/Container](https://en.cppreference.com/w/cpp/named_req/Container)) concept. In particular, iterators and associated operations (`begin`, `end`, etc.) are not provided. Element access is done through user-provided *visitation functions* that are passed to `concurrent_flat_set` operations where they are executed internally in a controlled fashion. Such visitation-based API allows for low-contention concurrent usage scenarios.

The internal data structure of `boost::concurrent_flat_set` is similar to that of `boost::unordered_flat_set`. As a result of its using open-addressing techniques, `value_type` must be move-constructible and pointer stability is not kept under rehashing.

## Synopsis

```

// #include <boost/unordered/concurrent_flat_set.hpp>

namespace boost {
    template<class Key,
              class Hash = boost::hash<Key>,
              class Pred = std::equal_to<Key>,
              class Allocator = std::allocator<Key>>
    class concurrent_flat_set {
public:
    // types
    using key_type          = Key;
    using value_type         = Key;
    using init_type          = Key;
    using hasher             = Hash;
    using key_equal          = Pred;
    using allocator_type     = Allocator;
    using pointer             = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer       = typename std::allocator_traits<Allocator>::const_pointer;
    using reference           = value_type&;
    using const_reference     = const value_type&;
    using size_type           = std::size_t;
    using difference_type     = std::ptrdiff_t;

    // constants
    static constexpr size_type bulk_visit_size = implementation-defined;

    // construct/copy/destroy
    concurrent_flat_set();
    explicit concurrent_flat_set(size_type n,
                                  const hasher& hf = hasher(),
                                  const key_equal& eql = key_equal(),
                                  const allocator_type& a = allocator_type());
    template<class InputIterator>
    concurrent_flat_set(InputIterator f, InputIterator l,
                        size_type n = implementation-defined,
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());
    concurrent_flat_set(const concurrent_flat_set& other);
    concurrent_flat_set(concurrent_flat_set&& other);
    template<class InputIterator>
    concurrent_flat_set(InputIterator f, InputIterator l, const allocator_type& a);
    explicit concurrent_flat_set(const Allocator& a);
    concurrent_flat_set(const concurrent_flat_set& other, const Allocator& a);
    concurrent_flat_set(concurrent_flat_set&& other, const Allocator& a);
    concurrent_flat_set(unordered_flat_set<Key, Hash, Pred, Allocator>&& other);
    concurrent_flat_set(std::initializer_list<value_type> il,
                        size_type n = implementation-defined
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());
    concurrent_flat_set(size_type n, const allocator_type& a);
    concurrent_flat_set(size_type n, const hasher& hf, const allocator_type& a);
    template<class InputIterator>
    concurrent_flat_set(InputIterator f, InputIterator l, size_type n,
                        const allocator_type& a);
    template<class InputIterator>
    concurrent_flat_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                        const allocator_type& a);
    concurrent_flat_set(std::initializer_list<value_type> il, const allocator_type& a);
    concurrent_flat_set(std::initializer_list<value_type> il, size_type n,
                        const allocator_type& a);
    concurrent_flat_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,

```

```

        const allocator_type& a);
~concurrent_flat_set();
concurrent_flat_set& operator=(const concurrent_flat_set& other);
concurrent_flat_set& operator=(concurrent_flat_set&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value || 
            boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value);
concurrent_flat_set& operator=(std::initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// visitation
template<class F> size_t visit(const key_type& k, F f) const;
template<class F> size_t cvisit(const key_type& k, F f) const;
template<class K, class F> size_t visit(const K& k, F f) const;
template<class K, class F> size_t cvisit(const K& k, F f) const;

template<class FwdIterator, class F>
    size_t visit(FwdIterator first, FwdIterator last, F f) const;
template<class FwdIterator, class F>
    size_t cvisit(FwdIterator first, FwdIterator last, F f) const;

template<class F> size_t visit_all(F f) const;
template<class F> size_t cvisit_all(F f) const;
template<class ExecutionPolicy, class F>
    void visit_all(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F>
    void cvisit_all(ExecutionPolicy&& policy, F f) const;

template<class F> bool visit_while(F f) const;
template<class F> bool cvisit_while(F f) const;
template<class ExecutionPolicy, class F>
    bool visit_while(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F>
    bool cvisit_while(ExecutionPolicy&& policy, F f) const;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> bool emplace(Args&&... args);
bool insert(const value_type& obj);
bool insert(value_type&& obj);
template<class K> bool insert(K&& k);
template<class InputIterator> size_type insert(InputIterator first, InputIterator last);
size_type insert(std::initializer_list<value_type> il);

template<class... Args, class F> bool emplace_or_visit(Args&&... args, F&& f);
template<class... Args, class F> bool emplace_or_cvisit(Args&&... args, F&& f);
template<class F> bool insert_or_visit(const value_type& obj, F f);
template<class F> bool insert_or_cvisit(const value_type& obj, F f);
template<class F> bool insert_or_visit(value_type&& obj, F f);
template<class F> bool insert_or_cvisit(value_type&& obj, F f);
template<class K, class F> bool insert_or_visit(K&& k, F f);
template<class K, class F> bool insert_or_cvisit(K&& k, F f);
template<class InputIterator, class F>
    size_type insert_or_visit(InputIterator first, InputIterator last, F f);
template<class InputIterator, class F>
    size_type insert_or_cvisit(InputIterator first, InputIterator last, F f);
template<class F> size_type insert_or_visit(std::initializer_list<value_type> il, F f);
template<class F> size_type insert_or_cvisit(std::initializer_list<value_type> il, F f);

```

```

size_type erase(const key_type& k);
template<class K> size_type erase(const K& k);

template<class F> size_type erase_if(const key_type& k, F f);
template<class K, class F> size_type erase_if(const K& k, F f);
template<class F> size_type erase_if(F f);
template<class ExecutionPolicy, class F> void erase_if(ExecutionPolicy&& policy, F f);

void swap(concurrent_flat_set& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
void clear() noexcept;

template<class H2, class P2>
    size_type merge(concurrent_flat_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
    size_type merge(concurrent_flat_set<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
size_type count(const key_type& k) const;
template<class K>
    size_type count(const K& k) const;
bool contains(const key_type& k) const;
template<class K>
    bool contains(const K& k) const;

// bucket interface
size_type bucket_count() const noexcept;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
size_type max_load() const noexcept;
void rehash(size_type n);
void reserve(size_type n);
};

// Deduction Guides
template<class InputIterator,
         class Hash = boost::hash<iter-value-type<InputIterator>>,
         class Pred = std::equal_to<iter-value-type<InputIterator>>,
         class Allocator = std::allocator<iter-value-type<InputIterator>>>
concurrent_flat_set(InputIterator, InputIterator, typename see below::size_type = see below,
                    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> concurrent_flat_set<iter-value-type<InputIterator>, Hash, Pred, Allocator>;

template<class T, class Hash = boost::hash<T>, class Pred = std::equal_to<T>,
         class Allocator = std::allocator<T>>
concurrent_flat_set(std::initializer_list<T>, typename see below::size_type = see below,
                    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> concurrent_flat_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
concurrent_flat_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> concurrent_flat_set<iter-value-type<InputIterator>,
               boost::hash<iter-value-type<InputIterator>>,
               std::equal_to<iter-value-type<InputIterator>>, Allocator>;

```

```

template<class InputIterator, class Allocator>
concurrent_flat_set(InputIterator, InputIterator, Allocator)
-> concurrent_flat_set<iter_value_type<InputIterator>,
    boost::hash<iter_value_type<InputIterator>>,
    std::equal_to<iter_value_type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
concurrent_flat_set(InputIterator, InputIterator, typename see below::size_type, Hash,
                    Allocator)
-> concurrent_flat_set<iter_value_type<InputIterator>, Hash,
    std::equal_to<iter_value_type<InputIterator>>, Allocator>;

template<class T, class Allocator>
concurrent_flat_set(std::initializer_list<T>, typename see below::size_type, Allocator)
-> concurrent_flat_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Allocator>
concurrent_flat_set(std::initializer_list<T>, Allocator)
-> concurrent_flat_set<T, boost::hash<T>, std::equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
concurrent_flat_set(std::initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> concurrent_flat_set<T, Hash, std::equal_to<T>, Allocator>;

// Equality Comparisons
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
                  const concurrent_flat_set<Key, Hash, Pred, Alloc>& y);

template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
                  const concurrent_flat_set<Key, Hash, Pred, Alloc>& y);

// swap
template<class Key, class Hash, class Pred, class Alloc>
void swap(concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
          concurrent_flat_set<Key, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

// Erasure
template<class K, class H, class P, class A, class Predicate>
typename concurrent_flat_set<K, H, P, A>::size_type
erase_if(concurrent_flat_set<K, H, P, A>& c, Predicate pred);
}

```

## Description

### Template Parameters

<i>Key</i>	Key must be <u><a href="#">MoveInsertable</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/MoveInsertable">https://en.cppreference.com/w/cpp/named_req/MoveInsertable</a> ) into the container and <u><a href="#">Erasable</a></u> ( <a href="https://en.cppreference.com/w/cpp/named_req/Erasable">https://en.cppreference.com/w/cpp/named_req/Erasable</a> ) from the container.
<i>Hash</i>	A unary function object type that acts a hash function for a <i>Key</i> . It takes a single argument of type <i>Key</i> and returns a value of type <i>std::size_t</i> .

<code>Pred</code>	A binary function object that induces an equivalence relation on values of type <code>Key</code> . It takes two arguments of type <code>Key</code> and returns a value of type <code>bool</code> .
<code>Allocator</code>	An allocator whose value type is the same as the table's value type. <code>std::allocator_traits&lt;Allocator&gt;::pointer</code> and <code>std::allocator_traits&lt;Allocator&gt;::const_pointer</code> must be convertible to/from <code>value_type*</code> and <code>const value_type*</code> , respectively.

The elements of the table are held into an internal *bucket array*. An element is inserted into a bucket determined by its hash code, but if the bucket is already occupied (a *collision*), an available one in the vicinity of the original position is used.

The size of the bucket array can be automatically increased by a call to `insert` / `emplace`, or as a result of calling `rehash` / `reserve`. The *load factor* of the table (number of elements divided by number of buckets) is never greater than `max_load_factor()`, except possibly for small sizes where the implementation may decide to allow for higher loads.

If `hash_is_avalanching<Hash>::value` is `true`, the hash function is used as-is; otherwise, a bit-mixing post-processing stage is added to increase the quality of hashing at the expense of extra computational cost.

## Concurrency Requirements and Guarantees

Concurrent invocations of `operator()` on the same `const` instance of `Hash` or `Pred` are required to not introduce data races. For `Alloc` being either `Allocator` or any allocator type rebound from `Allocator`, concurrent invocations of the following operations on the same instance `al` of `Alloc` are required to not introduce data races:

- Copy construction from `al` of an allocator rebound from `Alloc`
- `std::allocator_traits<Alloc>::allocate`
- `std::allocator_traits<Alloc>::deallocate`
- `std::allocator_traits<Alloc>::construct`
- `std::allocator_traits<Alloc>::destroy`

In general, these requirements on `Hash`, `Pred` and `Allocator` are met if these types are not stateful or if the operations only involve constant access to internal data members.

With the exception of destruction, concurrent invocations of any operation on the same instance of a `concurrent_flat_set` do not introduce data races — that is, they are thread-safe.

If an operation `op` is explicitly designated as *blocking on* `x`, where `x` is an instance of a `boost::concurrent_flat_set`, prior blocking operations on `x` synchronize with `op`. So, blocking operations on the same `concurrent_flat_set` execute sequentially in a multithreaded scenario.

An operation is said to be *blocking on rehashing of* `x` if it blocks on `x` only when an internal rehashing is issued.

Access or modification of an element of a `boost::concurrent_flat_set` passed by reference to a user-provided visitation function do not introduce data races when the visitation function is executed internally by the `boost::concurrent_flat_set`.

Any `boost::concurrent_flat_set` operation that inserts or modifies an element `e` synchronizes with the internal invocation of a visitation function on `e`.

Visitation functions executed by a `boost::concurrent_flat_set x` are not allowed to invoke any operation on `x`; invoking operations on a different `boost::concurrent_flat_set` instance `y` is allowed only if concurrent outstanding operations on `y` do not access `x` directly or indirectly.

## Configuration Macros

`BOOST_UNORDERED_DISABLE_REENTRANCY_CHECK`

In debug builds (more precisely, when `BOOST_ASSERT_IS_VOID` is not defined), *container reentrancies* (illegally invoking an operation on `m` from within a function visiting elements of `m`) are detected and signalled through `BOOST_ASSERT_MSG`. When run-time speed is a concern, the feature can be disabled by globally defining this macro.

## Constants

`static constexpr size_type bulk_visit_size;`

CPP

Chunk size internally used in bulk visit operations.

## Constructors

### Default Constructor

`concurrent_flat_set();`

C++

Constructs an empty table using `hasher()` as the hash function, `key_equal()` as the key equality predicate and `allocator_type()` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

### Bucket Count Constructor

`explicit concurrent_flat_set(size_type n,  
 const hasher& hf = hasher(),  
 const key_equal& eql = key_equal(),  
 const allocator_type& a = allocator_type());`

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate, and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor

```
template<class InputIterator>
concurrent_flat_set(InputIterator f, InputIterator l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

C++

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a` as the allocator, and inserts the elements from `[f, l)` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Copy Constructor

```
concurrent_flat_set(concurrent_flat_set const& other);
```

C++

The copy constructor. Copies the contained elements, hash function, predicate and allocator.

If `Allocator::select_on_container_copy_construction` exists and has the right signature, the allocator will be constructed from its result.

**Requires:** `value_type` is copy constructible

**Concurrency:** Blocking on `other`.

---

## Move Constructor

```
concurrent_flat_set(concurrent_flat_set&& other);
```

C++

The move constructor. The internal bucket array of `other` is transferred directly to the new table. The hash function, predicate and allocator are moved-constructed from `other`.

**Concurrency:** Blocking on `other`.

---

## Iterator Range Constructor with Allocator

```
template<class InputIterator>
concurrent_flat_set(InputIterator f, InputIterator l, const allocator_type& a);
```

Constructs an empty table using `a` as the allocator, with the default hash function and key equality predicate and inserts the elements from `[f, l)` into it.

**Requires:** `hasher`, `key_equal` need to be [DefaultConstructible](#)  
([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Allocator Constructor

```
explicit concurrent_flat_set(Allocator const& a);
```

Constructs an empty table, using allocator `a`.

---

## Copy Constructor with Allocator

```
concurrent_flat_set(concurrent_flat_set const& other, Allocator const& a);
```

Constructs a table, copying `other`'s contained elements, hash function, and predicate, but using allocator `a`.

**Concurrency:** Blocking on `other`.

---

## Move Constructor with Allocator

```
concurrent_flat_set(concurrent_flat_set&& other, Allocator const& a);
```

If `a == other.get_allocator()`, the elements of `other` are transferred directly to the new table; otherwise, elements are moved-constructed from those of `other`. The hash function and predicate are moved-constructed from `other`, and the allocator is copy-constructed from `a`.

**Concurrency:** Blocking on `other`.

---

## Move Constructor from `unordered_flat_set`

```
concurrent_flat_set(unordered_flat_set<Key, Hash, Pred, Allocator>&& other);
```

Move construction from a `unordered_flat_set`. The internal bucket array of `other` is transferred directly to the new container. The hash function, predicate and allocator are moved-constructed from `other`.

**Complexity:**  $O(\text{bucket\_count}())$

---

## Initializer List Constructor

C++

```
concurrent_flat_set(std::initializer_list<value_type> il,
    size_type n = implementation-defined,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, `eql` as the key equality predicate and `a`, and inserts the elements from `il` into it.

**Requires:** If the defaults are used, `hasher`, `key_equal` and `allocator_type` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Allocator

C++

```
concurrent_flat_set(size_type n, allocator_type const& a);
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, the default hash function and key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `hasher` and `key_equal` need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Bucket Count Constructor with Hasher and Allocator

C++

```
concurrent_flat_set(size_type n, hasher const& hf, allocator_type const& a);
```

Constructs an empty table with at least `n` buckets, using `hf` as the hash function, the default key equality predicate and `a` as the allocator.

**Postconditions:** `size() == 0`

**Requires:** `key_equal` needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Allocator

C++

```
template<class InputIterator>
concurrent_flat_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a);
```

Constructs an empty table with at least `n` buckets, using `a` as the allocator and default hash function and key equality predicate, and inserts the elements from `[f, l)` into it.

**Requires:** hasher , key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Iterator Range Constructor with Bucket Count and Hasher

C++

```
template<class InputIterator>
concurrent_flat_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                     const allocator_type& a);
```

Constructs an empty table with at least n buckets, using hf as the hash function, a as the allocator, with the default key equality predicate, and inserts the elements from [f, l) into it.

**Requires:** key\_equal needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Allocator

C++

```
concurrent_flat_set(std::initializer_list<value_type> il, const allocator_type& a);
```

Constructs an empty table using a and default hash function and key equality predicate, and inserts the elements from il into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Allocator

C++

```
concurrent_flat_set(std::initializer_list<value_type> il, size_type n, const allocator_type& a);
```

Constructs an empty table with at least n buckets, using a and default hash function and key equality predicate, and inserts the elements from il into it.

**Requires:** hasher and key\_equal need to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## initializer\_list Constructor with Bucket Count and Hasher and Allocator

C++

```
concurrent_flat_set(std::initializer_list<value_type> il, size_type n, const hasher& hf,
                     const allocator_type& a);
```

Constructs an empty table with at least n buckets, using hf as the hash function, a as the allocator and default key equality predicate, and inserts the elements from il into it.

**Requires:** key\_equal needs to be [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)).

---

## Destructor

```
~concurrent_flat_set();
```

C++

**Note:** The destructor is applied to every element, and all memory is deallocated

---

## Assignment

### Copy Assignment

```
concurrent_flat_set& operator=(concurrent_flat_set const& other);
```

C++

The assignment operator. Destroys previously existing elements, copy-assigns the hash function and predicate from `other`, copy-assigns the allocator from `other` if `Alloc::propagate_on_container_copy_assignment` exists and `Alloc::propagate_on_container_copy_assignment::value` is `true`, and finally inserts copies of the elements of `other`.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

**Concurrency:** Blocking on `*this` and `other`.

---

### Move Assignment

```
concurrent_flat_set& operator=(concurrent_flat_set&& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
              boost::allocator_traits<Allocator>::propagate_on_container_move_assignment::value);
```

C++

The move assignment operator. Destroys previously existing elements, swaps the hash function and predicate from `other`, and move-assigns the allocator from `other` if `Alloc::propagate_on_container_move_assignment` exists and `Alloc::propagate_on_container_move_assignment::value` is `true`. If at this point the allocator is equal to `other.get_allocator()`, the internal bucket array of `other` is transferred directly to `*this`; otherwise, inserts move-constructed copies of the elements of `other`.

**Concurrency:** Blocking on `*this` and `other`.

---

### Initializer List Assignment

```
concurrent_flat_set& operator=(std::initializer_list<value_type> il);
```

C++

Assign from values in initializer list. All previously existing elements are destroyed.

**Requires:** `value_type` is [CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable))

**Concurrency:** Blocking on `*this`.

---

## Visitation

### [c]visit

```
template<class F> size_t visit(const key_type& k, F f) const;
template<class F> size_t cvisit(const key_type& k, F f) const;
template<class K, class F> size_t visit(const K& k, F f) const;
template<class K, class F> size_t cvisit(const K& k, F f) const;
```

C++

If an element  $x$  exists with key equivalent to  $k$ , invokes  $f$  with a const reference to  $x$ .

**Returns:** The number of elements visited (0 or 1).

**Notes:** The `template<class K, class F>` overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Bulk visit

```
template<class FwdIterator, class F>
size_t visit(FwdIterator first, FwdIterator last, F f) const;
template<class FwdIterator, class F>
size_t cvisit(FwdIterator first, FwdIterator last, F f) const;
```

C++

For each element  $k$  in the range  $[first, last)$ , if there is an element  $x$  in the container with key equivalent to  $k$ , invokes  $f$  with a const reference to  $x$ .

Although functionally equivalent to individually invoking `[c]visit` for each key, bulk visitation performs generally faster due to internal streamlining optimizations. It is advisable that `std::distance(first, last)` be at least `bulk_visit_size` to enjoy a performance gain: beyond this size, performance is not expected to increase further.

**Requires:** `FwdIterator` is a [LegacyForwardIterator](https://en.cppreference.com/w/cpp/named_req/ForwardIterator) (C++11 to C++17), or satisfies [std::forward\\_iterator](https://en.cppreference.com/w/cpp/iterator/forward_iterator) (C++20 and later). For  $K = \text{std}\colon\text{iterator_traits}<\text{FwdIterator}\rangle\colon\text{value\_type}$ , either  $K$  is `key_type` or else `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. In the latter case, the library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

**Returns:** The number of elements visited.

---

### [c]visit\_all

```
template<class F> size_t visit_all(F f) const;
template<class F> size_t cvisit_all(F f) const;
```

C++

Successively invokes  $f$  with const references to each of the elements in the table.

**Returns:** The number of elements visited.

---

## Parallel [c]visit\_all

```
template<class ExecutionPolicy, class F> void visit_all(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F> void cvisit_all(ExecutionPolicy&& policy, F f) const;
```

C++

Invokes `f` with const references to each of the elements in the table. Execution is parallelized according to the semantics of the execution policy specified.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

These overloads only participate in overload resolution if

`std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true`.

Unsequenced execution policies are not allowed.

---

## [c]visit\_while

```
template<class F> bool visit_while(F f) const;
template<class F> bool cvisit_while(F f) const;
```

C++

Successively invokes `f` with const references to each of the elements in the table until `f` returns `false` or all the elements are visited.

**Returns:** `false` iff `f` ever returns `false`.

---

## Parallel [c]visit\_while

```
template<class ExecutionPolicy, class F> bool visit_while(ExecutionPolicy&& policy, F f) const;
template<class ExecutionPolicy, class F> bool cvisit_while(ExecutionPolicy&& policy, F f) const;
```

C++

Invokes `f` with const references to each of the elements in the table until `f` returns `false` or all the elements are visited. Execution is parallelized according to the semantics of the execution policy specified.

**Returns:** `false` iff `f` ever returns `false`.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

These overloads only participate in overload resolution if  
`std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true`.

Unsequenced execution policies are not allowed.

Parallelization implies that execution does not necessarily finish as soon as `f` returns `false`, and as a result `f` may be invoked with further elements for which the return value is also `false`.

---

## Size and Capacity

### empty

C++  
[[nodiscard]] `bool empty() const noexcept;`

**Returns:** `size() == 0`

---

### size

C++  
`size_type size() const noexcept;`

**Returns:** The number of elements in the table.

**Notes:** In the presence of concurrent insertion operations, the value returned may not accurately reflect the true size of the table right after execution.

---

### max\_size

C++  
`size_type max_size() const noexcept;`

**Returns:** `size()` of the largest possible table.

---

## Modifiers

### emplace

C++  
`template<class... Args> bool emplace(Args&&... args);`

Inserts an object, constructed with the arguments `args`, in the table if and only if there is no element in the table with an equivalent key.

**Requires:** `value_type` is constructible from `args`.

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Copy Insert

`bool insert(const value_type& obj);`

C++

Inserts obj in the table if and only if there is no element in the table with an equivalent key.

**Requires:** value\_type is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Move Insert

`bool insert(value_type&& obj);`

C++

Inserts obj in the table if and only if there is no element in the table with an equivalent key.

**Requires:** value\_type is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of \*this .

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Transparent Insert

`template<class K> bool insert(K&& k);`

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key.

**Requires:** value\_type is [EmplaceConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from k .

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

This overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert Iterator Range

```
template<class InputIterator> size_type insert(InputIterator first, InputIterator last);
```

C++

Equivalent to

```
while(first != last) this->emplace(*first++);
```

**Returns:** The number of elements inserted.

---

## Insert Initializer List

```
size_type insert(std::initializer_list<value_type> il);
```

C++

Equivalent to

```
this->insert(il.begin(), il.end());
```

**Returns:** The number of elements inserted.

---

## emplace\_or\_[c]visit

```
template<class... Args, class F> bool emplace_or_visit(Args&&... args, F&& f);
template<class... Args, class F> bool emplace_or_cvisit(Args&&... args, F&& f);
```

C++

Inserts an object, constructed with the arguments `args`, in the table if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a const reference to the equivalent element.

**Requires:** `value_type` is constructible from `args`.

**Returns:** `true` if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

The interface is exposition only, as C++ does not allow to declare a parameter `f` after a variadic parameter pack.

---

## Copy insert\_or\_[c]visit

```
template<class F> bool insert_or_visit(const value_type& obj, F f);
template<class F> bool insert_or_cvisit(const value_type& obj, F f);
```

C++

Inserts `obj` in the table if and only if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a const reference to the equivalent element.

**Requires:** `value_type` is [CopyInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/CopyInsertable](https://en.cppreference.com/w/cpp/named_req/CopyInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Move insert\_or\_[c]visit

```
template<class F> bool insert_or_visit(value_type&& obj, F f);
template<class F> bool insert_or_cvisit(value_type&& obj, F f);
```

C++

Inserts `obj` in the table if and only if there is no element in the table with an equivalent key. Otherwise, invokes `f` with a const reference to the equivalent element.

**Requires:** `value_type` is [MoveInsertable](#) ([https://en.cppreference.com/w/cpp/named\\_req/MoveInsertable](https://en.cppreference.com/w/cpp/named_req/MoveInsertable)).

**Returns:** true if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

---

## Transparent insert\_or\_[c]visit

```
template<class K, class F> bool insert_or_visit(K&& k, F f);
template<class K, class F> bool insert_or_cvisit(K&& k, F f);
```

C++

Inserts an element constructed from `std::forward<K>(k)` in the container if and only if there is no element in the container with an equivalent key. Otherwise, invokes `f` with a const reference to the equivalent element.

**Requires:** `value_type` is [EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible) ([https://en.cppreference.com/w/cpp/named\\_req/EmplaceConstructible](https://en.cppreference.com/w/cpp/named_req/EmplaceConstructible)) from `k`.

**Returns:** `true` if an insert took place.

**Concurrency:** Blocking on rehashing of `*this`.

**Notes:** Invalidates pointers and references to elements if a rehashing is issued.

These overloads only participate in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## Insert Iterator Range or Visit

```
template<class InputIterator, class F>
size_type insert_or_visit(InputIterator first, InputIterator last, F f);
template<class InputIterator, class F>
size_type insert_or_cvisit(InputIterator first, InputIterator last, F f);
```

C++

Equivalent to

```
while(first != last) this->emplace_or_[c]visit(*first++, f);
```

**Returns:** The number of elements inserted.

---

## Insert Initializer List or Visit

```
template<class F> size_type insert_or_visit(std::initializer_list<value_type> il, F f);
template<class F> size_type insert_or_cvisit(std::initializer_list<value_type> il, F f);
```

C++

Equivalent to

```
this->insert_or[c]visit(il.begin(), il.end(), f);
```

**Returns:** The number of elements inserted.

---

## erase

```
size_type erase(const key_type& k);
template<class K> size_type erase(const K& k);
```

C++

Erases the element with key equivalent to `k` if it exists.

**Returns:** The number of elements erased (0 or 1).

**Throws:** Only throws an exception if it is thrown by `hasher` or `key_equal`.

**Notes:** The template<class K> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## erase\_if by Key

C++

```
template<class F> size_type erase_if(const key_type& k, F f);
template<class K, class F> size_type erase_if(const K& k, F f);
```

Erases the element `x` with key equivalent to `k` if it exists and `f(x)` is `true`.

**Returns:** The number of elements erased (0 or 1).

**Throws:** Only throws an exception if it is thrown by `hasher`, `key_equal` or `f`.

**Notes:** The template<class K, class F> overload only participates in overload resolution if `std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `false`.

The template<class K, class F> overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

---

## erase\_if

C++

```
template<class F> size_type erase_if(F f);
```

Successively invokes `f` with references to each of the elements in the table, and erases those for which `f` returns `true`.

**Returns:** The number of elements erased.

**Throws:** Only throws an exception if it is thrown by `f`.

## Parallel erase\_if

C++

```
template<class ExecutionPolicy, class F> void erase_if(ExecutionPolicy&& policy, F f);
```

Invokes `f` with references to each of the elements in the table, and erases those for which `f` returns `true`. Execution is parallelized according to the semantics of the execution policy specified.

**Throws:** Depending on the exception handling mechanism of the execution policy used, may call `std::terminate` if an exception is thrown within `f`.

**Notes:** Only available in compilers supporting C++17 parallel algorithms.

This overload only participates in overload resolution if `std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>` is `true`.

Unsequenced execution policies are not allowed.

---

## swap

```
void swap(concurrent_flat_set& other)
    noexcept(boost::allocator_traits<Allocator>::is_always_equal::value ||
             boost::allocator_traits<Allocator>::propagate_on_container_swap::value);
```

C++

Swaps the contents of the table with the parameter.

If `Allocator::propagate_on_container_swap` is declared and `Allocator::propagate_on_container_swap::value` is `true` then the tables' allocators are swapped. Otherwise, swapping with unequal allocators results in undefined behavior.

**Throws:** Nothing unless `key_equal` or `hasher` throw on swapping.

**Concurrency:** Blocking on `*this` and `other`.

---

## clear

```
void clear() noexcept;
```

C++

Erases all elements in the table.

**Postconditions:** `size() == 0`, `max_load() >= max_load_factor() * bucket_count()`

**Concurrency:** Blocking on `*this`.

---

## merge

```
template<class H2, class P2>
    size_type merge(concurrent_flat_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
    size_type merge(concurrent_flat_set<Key, H2, P2, Allocator>&& source);
```

C++

Move-inserts all the elements from `source` whose key is not already present in `*this`, and erases them from `source`.

**Returns:** The number of elements inserted.

**Concurrency:** Blocking on `*this` and `source`.

---

## Observers

### get\_allocator

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

C++

**Returns:** The table's allocator.

---

### hash\_function

```
hasher hash_function() const;
```

C++

**Returns:** The table's hash function.

---

### key\_eq

```
key_equal key_eq() const;
```

C++

**Returns:** The table's key equality predicate.

---

## Set Operations

### count

```
size_type count(const key_type& k) const;  
template<class K>  
size_type count(const K& k) const;
```

C++

**Returns:** The number of elements with key equivalent to `k` (0 or 1).

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

### contains

```
bool contains(const key_type& k) const;
template<class K>
bool contains(const K& k) const;
```

**Returns:** A boolean indicating whether or not there is an element with key equal to `k` in the table.

**Notes:** The `template<class K>` overload only participates in overload resolution if `Hash::is_transparent` and `Pred::is_transparent` are valid member typedefs. The library assumes that `Hash` is callable with both `K` and `Key` and that `Pred` is transparent. This enables heterogeneous lookup which avoids the cost of instantiating an instance of the `Key` type.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

## Bucket Interface

### bucket\_count

```
size_type bucket_count() const noexcept;
```

**Returns:** The size of the bucket array.

---

## Hash Policy

### load\_factor

```
float load_factor() const noexcept;
```

**Returns:** `static_cast<float>(size()) / static_cast<float>(bucket_count())`, or 0 if `bucket_count() == 0`.

---

### max\_load\_factor

```
float max_load_factor() const noexcept;
```

**Returns:** Returns the table's maximum load factor.

---

## Set max\_load\_factor

```
void max_load_factor(float z);
```

**Effects:** Does nothing, as the user is not allowed to change this parameter. Kept for compatibility with `boost::unordered_set`.

---

## max\_load

C++

```
size_type max_load() const noexcept;
```

**Returns:** The maximum number of elements the table can hold without rehashing, assuming that no further elements will be erased.

**Note:** After construction, rehash or clearance, the table's maximum load is at least `max_load_factor() * bucket_count()`. This number may decrease on erasure under high-load conditions.

In the presence of concurrent insertion operations, the value returned may not accurately reflect the true state of the table right after execution.

---

## rehash

C++

```
void rehash(size_type n);
```

Changes if necessary the size of the bucket array so that there are at least `n` buckets, and so that the load factor is less than or equal to the maximum load factor. When applicable, this will either grow or shrink the `bucket_count()` associated with the table.

When `size() == 0`, `rehash(0)` will deallocate the underlying buckets array.

Invalidates pointers and references to elements, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the table's hash function or comparison function.

**Concurrency:** Blocking on `*this`.

---

## reserve

C++

```
void reserve(size_type n);
```

Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

Similar to `rehash`, this function can be used to grow or shrink the number of buckets in the table.

Invalidates pointers and references to elements, and changes the order of elements.

**Throws:** The function has no effect if an exception is thrown, unless it is thrown by the table's hash function or comparison function.

**Concurrency:** Blocking on `*this`.

---

A deduction guide will not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

A `size_type` parameter type in a deduction guide refers to the `size_type` member type of the container type deduced by the deduction guide. Its default value coincides with the default value of the constructor selected.

### *iter-value-type*

```
template<class InputIterator>
using iter-value-type =
    typename std::iterator_traits<InputIterator>::value_type; // exposition only
```

## Equality Comparisons

### operator==

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator==(const concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
                  const concurrent_flat_set<Key, Hash, Pred, Alloc>& y);
```

C++

Returns `true` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Concurrency:** Blocking on `x` and `y`.

**Notes:** Behavior is undefined if the two tables don't have equivalent equality predicates.

---

### operator!=

```
template<class Key, class Hash, class Pred, class Alloc>
bool operator!=(const concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
                  const concurrent_flat_set<Key, Hash, Pred, Alloc>& y);
```

C++

Returns `false` if `x.size() == y.size()` and for every element in `x`, there is an element in `y` with the same key, with an equal value (using `operator==` to compare the value types).

**Concurrency:** Blocking on `x` and `y`.

**Notes:** Behavior is undefined if the two tables don't have equivalent equality predicates.

---

## Swap

```
template<class Key, class Hash, class Pred, class Alloc>
void swap(concurrent_flat_set<Key, Hash, Pred, Alloc>& x,
          concurrent_flat_set<Key, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y))):
```

Equivalent to

```
x.swap(y);
```

## erase\_if

```
template<class K, class H, class P, class A, class Predicate>
typename concurrent_flat_set<K, H, P, A>::size_type
erase_if(concurrent_flat_set<K, H, P, A>& c, Predicate pred);
```

Equivalent to

```
c.erase_if(pred);
```

## Serialization

`concurrent_flat_set`s can be archived/retrieved by means of [Boost.Serialization](#) using the API provided by this library. Both regular and XML archives are supported.

### Saving an `concurrent_flat_set` to an archive

Saves all the elements of a `concurrent_flat_set` `x` to an archive (XML archive) `ar`.

**Requires:** `value_type` is serializable (XML serializable), and it supports Boost.Serialization `save_construct_data` / `load_construct_data` protocol (automatically supported by [DefaultConstructible](#) ([https://en.cppreference.com/w/cpp/named\\_req/DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible)) types).

**Concurrency:** Blocking on `x`.

### Loading an `concurrent_flat_set` from an archive

Deletes all preexisting elements of a `concurrent_flat_set` `x` and inserts from an archive (XML archive) `ar` restored copies of the elements of the original `concurrent_flat_set` `other` saved to the storage read by `ar`.

**Requires:** `x.key_equal()` is functionally equivalent to `other.key_equal()`.

**Concurrency:** Blocking on `x`.

## Change Log

### Release 1.84.0 - Major update

- Added `boost::concurrent_flat_set`.

- Added `[c]visit_while` operations to concurrent containers, with serial and parallel variants.
- Added efficient move construction of `boost::unordered_flat_(map|set)` from `boost::concurrent_flat_(map|set)` and vice versa.
- Added bulk visitation to concurrent containers for increased lookup performance.
- Added debug-mode mechanisms for detecting illegal reentrancies into a concurrent container from user code.
- Added Boost.Serialization support to all containers and their (non-local) iterator types.
- Added support for fancy pointers to open-addressing and concurrent containers. This enables scenarios like the use of Boost.Interprocess allocators to construct containers in shared memory.
- Fixed bug in member of pointer operator for local iterators of closed-addressing containers ([PR#221](#) (<https://github.com/boostorg/unordered/pull/221>), credit goes to GitHub user vslashg for finding and fixing this issue).
- Starting with this release, `boost::unordered_[multi]set` and `boost::unordered_[multi]map` only work with C++11 onwards.

## Release 1.83.0 - Major update

- Added `boost::concurrent_flat_map`, a fast, thread-safe hashmap based on open addressing.
- Sped up iteration of open-addressing containers.
- In open-addressing containers, `erase(iterator)`, which previously returned nothing, now returns a proxy object convertible to an iterator to the next element. This enables the typical `it = c.erase(it)` idiom without incurring any performance penalty when the returned proxy is not used.

## Release 1.82.0 - Major update

- C++03 support is planned for deprecation. Boost 1.84.0 will no longer support C++03 mode and C++11 will become the new minimum for using the library.
- Added node-based, open-addressing containers `boost::unordered_node_map` and `boost::unordered_node_set`.
- Extended heterogeneous lookup to more member functions as specified in [P2363](#) (<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2363r5.html>).
- Replaced the previous post-mixing process for open-addressing containers with a new algorithm based on extended multiplication by a constant.
- Fixed bug in internal `emplace()` impl where stack-local types were not properly constructed using the Allocator of the container which breaks uses-allocator construction.

## Release 1.81.0 - Major update

- Added fast containers `boost::unordered_flat_map` and `boost::unordered_flat_set` based on open addressing.
- Added CTAD deduction guides for all containers.
- Added missing constructors as specified in [LWG issue 2713](#) (<https://cplusplus.github.io/LWG/issue2713>).

## Release 1.80.0 - Major update

- Refactor internal implementation to be dramatically faster
- Allow `final` Hasher and KeyEqual objects

- Update documentation, adding benchmark graphs and notes on the new internal data structures

## Release 1.79.0

- Improved C++20 support:
  - All containers have been updated to support heterogeneous `count`, `equal_range` and `find`.
  - All containers now implement the member function `contains`.
  - `erase_if` has been implemented for all containers.
- Improved C++23 support:
  - All containers have been updated to support heterogeneous `erase` and `extract`.
- Changed behavior of `reserve` to eagerly allocate ([PR#59](https://github.com/boostorg/unordered/pull/59) (<https://github.com/boostorg/unordered/pull/59>)).
- Various warning fixes in the test suite.
- Update code to internally use `boost::allocator_traits`.
- Switch to Fibonacci hashing.
- Update documentation to be written in AsciiDoc instead of QuickBook.

## Release 1.67.0

- Improved C++17 support:
  - Add template deduction guides from the standard.
  - Use a simple implementation of `optional` in node handles, so that they're closer to the standard.
  - Add missing `noexcept` specifications to `swap`, `operator=` and node handles, and change the implementation to match. Using `std::allocator_traits::is_always_equal`, or our own implementation when not available, and `boost::is_nothrow_swappable` in the implementation.
- Improved C++20 support:
  - Use `boost::to_address`, which has the proposed C++20 semantics, rather than the old custom implementation.
  - Add `element_type` to iterators, so that `std::pointer_traits` will work.
  - Use `std::piecewise_construct` on recent versions of Visual C++, and other uses of the Dinkumware standard library, now using Boost.Predef to check compiler and library versions.
  - Use `std::iterator_traits` rather than the boost iterator traits in order to remove dependency on Boost.Iterator.
  - Remove iterators' inheritance from `std::iterator`, which is deprecated in C++17, thanks to Daniela Engert ([PR#7](https://github.com/boostorg/unordered/pull/7) (<https://github.com/boostorg/unordered/pull/7>)).
  - Stop using `BOOST_DEDUCED_TYPENAME`.
  - Update some Boost include paths.
  - Rename some internal methods, and variables.
  - Various testing improvements.
  - Miscellaneous internal changes.

## Release 1.66.0

- Simpler move construction implementation.
- Documentation fixes ([GitHub #6](https://github.com/boostorg/unordered/pull/6) (<https://github.com/boostorg/unordered/pull/6>)).

## Release 1.65.0

- Add deprecated attributes to `quick_erase` and `erase_return_void`. I really will remove them in a future version this time.
- Small standards compliance fixes:
  - `noexcept` specs for `swap` free functions.
  - Add missing `insert(P&&)` methods.

## Release 1.64.0

- Initial support for new C++17 member functions: `insert_or_assign` and `try_emplace` in `unordered_map`,
- Initial support for `merge` and `extract`. Does not include transferring nodes between `unordered_map` and `unordered_multimap` or between `unordered_set` and `unordered_multiset` yet. That will hopefully be in the next version of Boost.

## Release 1.63.0

- Check hint iterator in `insert / emplace_hint`.
- Fix some warnings, mostly in the tests.
- Manually write out `emplace_args` for small numbers of arguments - should make template error messages a little more bearable.
- Remove superfluous use of `boost::forward` in emplace arguments, which fixes emplacing string literals in old versions of Visual C++.
- Fix an exception safety issue in assignment. If bucket allocation throws an exception, it can overwrite the hash and equality functions while leaving the existing elements in place. This would mean that the function objects wouldn't match the container elements, so elements might be in the wrong bucket and equivalent elements would be incorrectly handled.
- Various reference documentation improvements.
- Better allocator support ([#12459](#) (<https://svn.boost.org/trac/boost/ticket/12459>)).
- Make the no argument constructors implicit.
- Implement missing allocator aware constructors.
- Fix assigning the hash/key equality functions for empty containers.
- Remove unary/binary\_function from the examples in the documentation. They are removed in C++17.
- Support 10 constructor arguments in `emplace`. It was meant to support up to 10 arguments, but an off by one error in the preprocessor code meant it only supported up to 9.

## Release 1.62.0

- Remove use of deprecated `boost::iterator`.

- Remove `BOOST_NO_STD_DISTANCE` workaround.
- Remove `BOOST_UNORDERED_DEPRECATED_EQUALITY` warning.
- Simpler implementation of assignment, fixes an exception safety issue for `unordered_multiset` and `unordered_multimap`. Might be a little slower.
- Stop using return value SFINAE which some older compilers have issues with.

## Release 1.58.0

- Remove unnecessary template parameter from `const` iterators.
- Rename `private iterator` `typedef` in some iterator classes, as it confuses some traits classes.
- Fix move assignment with stateful, `propagate_on_container_move_assign` allocators ([#10777](https://svn.boost.org/trac/boost/ticket/10777) (<https://svn.boost.org/trac/boost/ticket/10777>)).
- Fix rare exception safety issue in move assignment.
- Fix potential overflow when calculating number of buckets to allocate ([GitHub #4](#) (<https://github.com/boostorg/unordered/pull/4>)).

## Release 1.57.0

- Fix the `pointer` `typedef` in iterators ([#10672](#) (<https://svn.boost.org/trac/boost/ticket/10672>)).
- Fix Coverity warning ([GitHub #2](#) (<https://github.com/boostorg/unordered/pull/2>)).

## Release 1.56.0

- Fix some shadowed variable warnings ([#9377](#) (<https://svn.boost.org/trac/boost/ticket/9377>)).
- Fix allocator use in documentation ([#9719](#) (<https://svn.boost.org/trac/boost/ticket/9719>)).
- Always use prime number of buckets for integers. Fixes performance regression when inserting consecutive integers, although makes other uses slower ([#9282](#) (<https://svn.boost.org/trac/boost/ticket/9282>))).
- Only construct elements using allocators, as specified in C++11 standard.

## Release 1.55.0

- Avoid some warnings ([#8851](#) (<https://svn.boost.org/trac/boost/ticket/8851>)), [#8874](#) (<https://svn.boost.org/trac/boost/ticket/8874>)).
- Avoid exposing some detail functions via ADL on the iterators.
- Follow the standard by only using the allocators' construct and destroy methods to construct and destroy stored elements. Don't use them for internal data like pointers.

## Release 1.54.0

- Mark methods specified in standard as `noexcept`. More to come in the next release.
- If the hash function and equality predicate are known to both have `nothrow` move assignment or construction then use them.

## Release 1.53.0

- Remove support for the old pre-standard variadic pair constructors, and equality implementation. Both have been deprecated since Boost 1.48.
- Remove use of deprecated config macros.
- More internal implementation changes, including a much simpler implementation of `erase`.

## Release 1.52.0

- Faster assign, which assigns to existing nodes where possible, rather than creating entirely new nodes and copy constructing.
- Fixed bug in `erase_range` ([#7471](https://svn.boost.org/trac/boost/ticket/7471) (<https://svn.boost.org/trac/boost/ticket/7471>)).
- Reverted some of the internal changes to how nodes are created, especially for C++11 compilers. 'construct' and 'destroy' should work a little better for C++11 allocators.
- Simplified the implementation a bit. Hopefully more robust.

## Release 1.51.0

- Fix construction/destruction issue when using a C++11 compiler with a C++03 allocator ([#7100](https://svn.boost.org/trac/boost/ticket/7100) (<https://svn.boost.org/trac/boost/ticket/7100>)).
- Remove a `try..catch` to support compiling without exceptions.
- Adjust SFINAE use to try to support g++ 3.4 ([#7175](https://svn.boost.org/trac/boost/ticket/7175) (<https://svn.boost.org/trac/boost/ticket/7175>)).
- Updated to use the new config macros.

## Release 1.50.0

- Fix equality for `unordered_multiset` and `unordered_multimap`.
- [Ticket 6857](https://svn.boost.org/trac/boost/ticket/6857) (<https://svn.boost.org/trac/boost/ticket/6857>): Implement `reserve`.
- [Ticket 6771](https://svn.boost.org/trac/boost/ticket/6771) (<https://svn.boost.org/trac/boost/ticket/6771>): Avoid gcc's `-Wfloat-equal` warning.
- [Ticket 6784](https://svn.boost.org/trac/boost/ticket/6784) (<https://svn.boost.org/trac/boost/ticket/6784>): Fix some Sun specific code.
- [Ticket 6190](https://svn.boost.org/trac/boost/ticket/6190) (<https://svn.boost.org/trac/boost/ticket/6190>): Avoid gcc's `-Wshadow` warning.
- [Ticket 6905](https://svn.boost.org/trac/boost/ticket/6905) (<https://svn.boost.org/trac/boost/ticket/6905>): Make namespaces in macros compatible with `bcp` custom namespaces. Fixed by Luke Elliott.
- Remove some of the smaller prime number of buckets, as they may make collisions quite probable (e.g. multiples of 5 are very common because we used base 10).
- On old versions of Visual C++, use the container library's implementation of `allocator_traits`, as it's more likely to work.
- On machines with 64 bit `std::size_t`, use power of 2 buckets, with Thomas Wang's hash function to pick which one to use. As modulus is very slow for 64 bit values.
- Some internal changes.

## Release 1.49.0

- Fix warning due to accidental odd assignment.
- Slightly better error messages.

## Release 1.48.0 - Major update

This is major change which has been converted to use Boost.Move's move emulation, and be more compliant with the C++11 standard. See the compliance section for details.

The container now meets C++11's complexity requirements, but to do so uses a little more memory. This means that `quick_erase` and `erase_return_void` are no longer required, they'll be removed in a future version.

C++11 support has resulted in some breaking changes:

- Equality comparison has been changed to the C++11 specification. In a container with equivalent keys, elements in a group with equal keys used to have to be in the same order to be considered equal, now they can be a permutation of each other. To use the old behavior define the macro `BOOST_UNORDERED_DEPRECATED_EQUALITY`.
- The behaviour of `swap` is different when the two containers to be swapped has unequal allocators. It used to allocate new nodes using the appropriate allocators, it now swaps the allocators if the allocator has a member structure `propagate_on_container_swap`, such that `propagate_on_container_swap::value` is true.
- Allocator's `construct` and `destroy` functions are called with raw pointers, rather than the allocator's `pointer` type.
- `emplace` used to emulate the variadic pair constructors that appeared in early C++0x drafts. Since they were removed it no longer does so. It does emulate the new `piecewise_construct` pair constructors - only you need to use `boost::piecewise_construct`. To use the old emulation of the variadic constructors define `BOOST_UNORDERED_DEPRECATED_PAIR_CONSTRUCT`.

## Release 1.45.0

- Fix a bug when inserting into an `unordered_map` or `unordered_set` using iterators which returns `value_type` by copy.

## Release 1.43.0

- [Ticket 3966](https://svn.boost.org/trac/boost/ticket/3966) (<https://svn.boost.org/trac/boost/ticket/3966>): `erase_return_void` is now `quick_erase`, which is the [current forerunner for resolving the slow erase by iterator](#) ([http://home.roadrunner.com/~hinnant/issue\\_review/lwg-active.html#579](http://home.roadrunner.com/~hinnant/issue_review/lwg-active.html#579)), although there's a strong possibility that this may change in the future. The old method name remains for backwards compatibility but is considered deprecated and will be removed in a future release.
- Use Boost.Exception.
- Stop using deprecated `BOOST_HAS_*` macros.

## Release 1.42.0

- Support instantiating the containers with incomplete value types.
- Reduced the number of warnings (mostly in tests).
- Improved codegear compatibility.

- [Ticket 3693](https://svn.boost.org/trac/boost/ticket/3693) (https://svn.boost.org/trac/boost/ticket/3693): Add `erase_return_void` as a temporary workaround for the current `erase` which can be inefficient because it has to find the next element to return an iterator.
- Add templated find overload for compatible keys.
- [Ticket 3773](https://svn.boost.org/trac/boost/ticket/3773) (https://svn.boost.org/trac/boost/ticket/3773): Add missing `std` qualifier to `ptrdiff_t`.
- Some code formatting changes to fit almost all lines into 80 characters.

## Release 1.41.0 - Major update

- The original version made heavy use of macros to sidestep some of the older compilers' poor template support. But since I no longer support those compilers and the macro use was starting to become a maintenance burden it has been rewritten to use templates instead of macros for the implementation classes.
- The container object is now smaller thanks to using `boost::compressed_pair` for EBO and a slightly different function buffer - now using a `bool` instead of a member pointer.
- Buckets are allocated lazily which means that constructing an empty container will not allocate any memory.

## Release 1.40.0

- [Ticket 2975](https://svn.boost.org/trac/boost/ticket/2975) (https://svn.boost.org/trac/boost/ticket/2975): Store the prime list as a preprocessor sequence - so that it will always get the length right if it changes again in the future.
- [Ticket 1978](https://svn.boost.org/trac/boost/ticket/1978) (https://svn.boost.org/trac/boost/ticket/1978): Implement `emplace` for all compilers.
- [Ticket 2908](https://svn.boost.org/trac/boost/ticket/2908) (https://svn.boost.org/trac/boost/ticket/2908), [Ticket 3096](https://svn.boost.org/trac/boost/ticket/3096) (https://svn.boost.org/trac/boost/ticket/3096): Some workarounds for old versions of borland, including adding explicit destructors to all containers.
- [Ticket 3082](https://svn.boost.org/trac/boost/ticket/3082) (https://svn.boost.org/trac/boost/ticket/3082): Disable incorrect Visual C++ warnings.
- Better configuration for C++0x features when the headers aren't available.
- Create less buckets by default.

## Release 1.39.0

- [Ticket 2756](https://svn.boost.org/trac/boost/ticket/2756) (https://svn.boost.org/trac/boost/ticket/2756): Avoid a warning on Visual C++ 2009.
- Some other minor internal changes to the implementation, tests and documentation.
- Avoid an unnecessary copy in `operator[]`.
- [Ticket 2975](https://svn.boost.org/trac/boost/ticket/2975) (https://svn.boost.org/trac/boost/ticket/2975): Fix length of prime number list.

## Release 1.38.0

- Use `boost::swap`.
- [Ticket 2237](https://svn.boost.org/trac/boost/ticket/2237) (https://svn.boost.org/trac/boost/ticket/2237): Document that the equality and inequality operators are undefined for two objects if their equality predicates aren't equivalent. Thanks to Daniel Krügler.
- [Ticket 1710](https://svn.boost.org/trac/boost/ticket/1710) (https://svn.boost.org/trac/boost/ticket/1710): Use a larger prime number list. Thanks to Thorsten Ottosen and Hervé Brönnimann.
- Use `aligned_storage` to store the types. This changes the way the allocator is used to construct nodes. It used to construct the node with two calls to the allocator's `construct` method - once for the pointers and once for the value. It now constructs the node with a single call to construct and then constructs the value using in place construction.

- Add support for C++0x initializer lists where they're available (currently only g++ 4.4 in C++0x mode).

## Release 1.37.0

- Rename overload of `emplace` with hint, to `emplace_hint` as specified in [n2691](#) (<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2691.pdf>).
- Provide forwarding headers at `<boost/unordered/unordered_map_fwd.hpp>` and `<boost/unordered/unordered_set_fwd.hpp>`.
- Move all the implementation inside `boost/unordered`, to assist modularization and hopefully make it easier to track Release subversion.

## Release 1.36.0

First official release.

- Rearrange the internals.
- Move semantics - full support when rvalue references are available, emulated using a cut down version of the Adobe move library when they are not.
- Emplace support when rvalue references and variadic template are available.
- More efficient node allocation when rvalue references and variadic template are available.
- Added equality operators.

## Boost 1.35.0 Add-on - 31st March 2008

Unofficial release uploaded to vault, to be used with Boost 1.35.0. Incorporated many of the suggestions from the review.

- Improved portability thanks to Boost regression testing.
- Fix lots of typos, and clearer text in the documentation.
- Fix floating point to `std::size_t` conversion when calculating sizes from the max load factor, and use `double` in the calculation for greater accuracy.
- Fix some errors in the examples.

## Review Version

Initial review version, for the review conducted from 7th December 2007 to 16th December 2007.

## Bibliography

- *C/C++ Users Journal*. February, 2006. Pete Becker. [STL and TR1: Part III - Unordered containers](#) (<http://www.ddj.com/cpp/184402066>).  
An introduction to the standard unordered containers.
- *Wikipedia*. [Hash table](#) ([https://en.wikipedia.org/wiki/Hash\\_table](https://en.wikipedia.org/wiki/Hash_table)).  
An introduction to hash table implementations. Discusses the differences between closed-addressing and open-addressing approaches.
- Peter Dimov, 2022. [Development Plan for Boost.Unordered](#) ([https://pdimov.github.io/articles/unordered\\_dev\\_plan.html](https://pdimov.github.io/articles/unordered_dev_plan.html)).

## Copyright and License

**Daniel James**

Copyright © 2003, 2004 Jeremy B. Maitin-Shepard

Copyright © 2005-2008 Daniel James

Copyright © 2022-2023 Christian Mazakas

Copyright © 2022-2023 Joaquín M López Muñoz

Copyright © 2022-2023 Peter Dimov

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at  
[http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))