
Boost.Unordered

Daniel James

Copyright © 2003, 2004 Jeremy B. Maitin-Shepard

Copyright © 2005-2008 Daniel James

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)

Table of Contents

Introduction	1
The Data Structure	2
Equality Predicates and Hash Functions	5
Comparison with Associative Containers	8
Implementation Rationale	9
Change Log	11
Reference	13
Bibliography	57

Introduction

For accessing data based on key lookup, the C++ standard library offers `std::set`, `std::map`, `std::multiset` and `std::multimap`. These are generally implemented using balanced binary trees so that lookup time has logarithmic complexity. That is generally okay, but in many cases a [hash table](#) can perform better, as accessing data has constant complexity, on average. The worst case complexity is linear, but that occurs rarely and with some care, can be avoided.

Also, the existing containers require a 'less than' comparison object to order their elements. For some data types this is impossible to implement or isn't practical. In contrast, a hash table only needs an equality function and a hash function for the key.

With this in mind, the [C++ Standard Library Technical Report](#) introduced the unordered associative containers, which are implemented using hash tables, and they have now been added to the [Working Draft of the C++ Standard](#).

This library supplies an almost complete implementation of the specification in the [Working Draft of the C++ Standard](#).

`unordered_set` and `unordered_multiset` are defined in the header `<boost/unordered_set.hpp>`

```
namespace boost {
    template <
        class Key,
        class Hash = boost::hash<Key>,
        class Pred = std::equal_to<Key>,
        class Alloc = std::allocator<Key> >
        class unordered_set;

    template<
        class Key,
        class Hash = boost::hash<Key>,
        class Pred = std::equal_to<Key>,
        class Alloc = std::allocator<Key> >
        class unordered_multiset;
}
```

`unordered_map` and `unordered_multimap` are defined in the header `<boost/unordered_map.hpp>`

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

    template<
        class Key, class Mapped,
        class Hash = boost::hash<Key>,
        class Pred = std::equal_to<Key>,
        class Alloc = std::allocator<Key> >
        class unordered_multimap;
}
```

When using Boost.TR1, these classes are included from `<unordered_set>` and `<unordered_map>`, with the classes added to the `std::tr1` namespace.

The containers are used in a similar manner to the normal associative containers:

```
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());
```

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

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

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

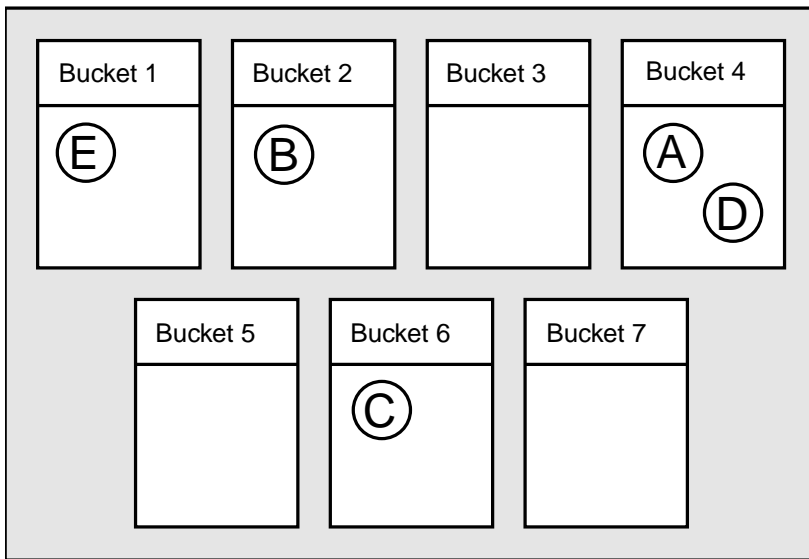
```
two,2
one,1
three,3
```

To store an object in an unordered associative container requires both an 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.

There are other differences, which are listed in the [Comparison with Associative Containers](#) section.

The Data Structure

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 an `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 `unordered_set` and `unordered_multiset` 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 that 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.

Table 1. Methods for Accessing Buckets

Method	Description
<code>size_type bucket_count() const</code>	The number of buckets.
<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 <i>n</i> .
<code>size_type bucket(key_type const& k) const</code>	Returns the index of the bucket which would contain <i>k</i>
<code>local_iterator begin(size_type n);</code>	Return begin and end iterators for bucket <i>n</i> .
<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>	
<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 elements in the buckets 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 are chosen, but it does make some requirements based on the container's 'load factor', the average number of elements per bucket. 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 draft 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 2. Methods for Controlling Bucket Size

Method	Description
<code>X(size_type n)</code>	Construct an empty container with at least <code>n</code> buckets (<code>x</code> is the container type).
<code>X(InputIterator i, InputIterator j, size_type n)</code>	Construct an empty container with at least <code>n</code> buckets and insert elements from the range <code>[i, j)</code> (<code>x</code> is the container type).
<code>float load_factor() const</code>	The average number of elements per bucket.
<code>float max_load_factor() const</code>	Returns the current maximum load factor.
<code>float max_load_factor(float z)</code>	Changes the container's maximum load factor, using <code>z</code> as a hint.
<code>void rehash(size_type n)</code>	Changes the number of buckets so that there at least <code>n</code> buckets, and so that the load factor is less than the maximum load factor.

Iterator Invalidation

It is not specified how member functions other than `rehash` affect the bucket count, although `insert` is only allowed to invalidate iterators when the insertion causes the load factor to be greater than or equal to the maximum load factor. For most implementations this means that `insert` will only change the number of buckets when this happens. While iterators can be invalidated by calls to `insert` and `rehash`, pointers and references to the container's elements are never invalidated.

In a similar manner to using `reserve` for vectors, it can be a good idea to call `rehash` 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.rehash((x.size() + n) / x.max_load_factor() + 1);
```

Note: `rehash`'s argument is the minimum number of buckets, not the number of elements, which is why the new size is divided by the maximum load factor. The `+ 1` guarantees there is no invalidation; without it, reallocation could occur if the number of bucket exactly divides the target size, since the container is allowed to rehash when the load factor is equal to the maximum load factor.

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;
```

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-1 hash](#) you could write:

```
boost::unordered_map<std::string, int, hash::fnv_1>
    dictionary;
```

There is an [implementation of FNV-1](#) 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:

```
struct iequal_to
    : std::binary_function<std::string, std::string, bool>
{
    bool operator()(std::string const& x,
        std::string const& y) const
    {
        return boost::algorithm::iequals(x, y, std::locale());
    }
};

struct ihash
    : std::unary_function<std::string, std::size_t>
{
    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:

```
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::unary_function<point, std::size_t>
{
    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 draft standard - so it won't work on other implementations of the unordered associative containers.

Table 3. Methods for accessing the hash and equality functions.

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

Comparison with Associative Containers

Table 4. 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 compared 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) && !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.
<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. 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 . Pointers and references to the container's elements are never invalidated.
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	Local iterators can be used to iterate through individual buckets. (I don't think that the order of local iterators and iterators are required to have any correspondence.)
Can be compared using the <code>==</code> , <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> operators.	No comparison operators are defined in the standard, although implementations might extend the containers to support <code>==</code> and <code>!=</code> .
	When inserting with a hint, implementations are permitted to ignore the hint.
<code>erase</code> never throws an exception	The containers' hash or predicate function can throw exceptions from <code>erase</code>

Table 5. 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 $O(N^2)$
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, upper_bound</code>	logarithmic	n/a

Implementation Rationale

The intent of this library is to implement the unordered containers in the draft standard, 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](#) 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 chained 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 draft 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 chained addressing, users will be surprised if the performance doesn't reflect that.

So chained addressing is used.

For containers with unique keys I store the buckets in a single-linked list. There are other possible data structures (such as a double-linked list) that allow for some operations to be faster (such as erasing and iteration) but the possible gain seems small compared to the extra memory needed. The most commonly used operations (insertion and lookup) would not be improved at all.

But for containers with equivalent keys a single-linked list can degrade badly when a large number of elements with equivalent keys are inserted. I think it's reasonable to assume that users who choose to use `unordered_multiset` or `unordered_multimap` do so because they are likely to insert elements with equivalent keys. So I have used an alternative data structure that doesn't degrade, at the expense of an extra pointer per node.

This works by adding storing a circular linked list for each group of equivalent nodes in reverse order. This allows quick navigation to the end of a group (since the first element points to the last) and can be quickly updated when elements are inserted or erased. The main disadvantage of this approach is some hairy code for erasing elements.

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.

Using a power of 2 allows for much quicker selection of the bucket to use, but at the expense of loosing 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](#). Unfortunately, a transformation like Wang's requires knowledge of the number of bits in the hash value, so it isn't portable enough. This leaves more expensive methods, such as Knuth's Multiplicative Method (mentioned in Wang's article). These don't tend to work as well as taking the modulus of a prime, and the extra computation required might negate efficiency advantage of power of 2 hash tables.

So, this implementation uses a prime number for the hash table size.

Equality operators

`operator==` and `operator!=` are not included in the standard, but I've added them as I think they could be useful and can be implemented fairly efficiently. They are specified differently to the other standard containers, comparing keys using the equality predicate rather than `operator==`.

It's also different to the proposal [n2944](#), which uses the equality operators for the whole of `value_type`. This implementation just uses the key equality function for the key, and `mapped_type`'s equality operator in `unordered_map` and `unordered_multimap` for the mapped part of the element.

Also, in `unordered_multimap`, the mapped values for a group of elements with equivalent keys are only considered equal if they are in the same order, in [n2944](#) they just need to be a permutation of each other. Since the order of elements with equal keys is now defined to be stable, it seems to me that their order can be considered part of the container's value.

Active Issues and Proposals

C++0x allocators

Recent drafts have included an overhaul of the allocators, but this was dependent on concepts which are no longer in the standard. [n2946](#) attempts to respecify them without concepts. I'll try to implement this (or an appropriate later version) in a future version of boost, possibly changed a little to accomodate non-C++0x compilers.

Swapping containers with unequal allocators

It isn't clear how to swap containers when their allocators aren't equal. This is [Issue 431: Swapping containers with unequal allocators](#). This has been resolved with the new allocator specification, so this should be fixed when support is added.

Are insert and erase stable for unordered_multiset and unordered_multimap?

It wasn't specified if `unordered_multiset` and `unordered_multimap` preserve the order of elements with equivalent keys (i.e. if they're stable under `insert` and `erase`). Since [n2691](#) it's been specified that they do and this implementation follows that.

Change Log

Review Version

Initial review version, for the review conducted from 7th December 2007 to 16th December 2007.

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.

Boost 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 1.37.0

- Rename overload of `emplace` with hint, to `emplace_hint` as specified in [n2691](#).

- 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 changes in subversion.

Boost 1.38.0

- Use `boost::swap`.
- [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](#): 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).

Boost 1.39.0

- [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](#): Fix length of prime number list.

Boost 1.40.0

- [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](#): Implement `emplace` for all compilers.
- [Ticket 2908](#), [Ticket 3096](#): Some workarounds for old versions of borland, including adding explicit destructors to all containers.
- [Ticket 3082](#): Disable incorrect Visual C++ warnings.
- Better configuration for C++0x features when the headers aren't available.
- Create less buckets by default.

Boost 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.

Boost 1.42.0

- Support instantiating the containers with incomplete value types.
- Reduced the number of warnings (mostly in tests).
- Improved codegear compatibility.
- [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](#): Add missing `std` qualifier to `ptrdiff_t`.
- Some code formatting changes to fit almost all lines into 80 characters.

Boost 1.43.0

- [Ticket 3966](#): `erase_return_void` is now `quick_erase`, which is the [current forerunner for resolving the slow erase by iterator](#), 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.

Reference

Header `<boost/unordered_set.hpp>`

```
namespace boost {
    template<typename Value, typename Hash = boost::hash<Value>,
            typename Pred = std::equal_to<Value>,
            typename Alloc = std::allocator<Value> >
        class unordered_set;
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        bool operator==(unordered_set<Value, Hash, Pred, Alloc> const&,
            unordered_set<Value, Hash, Pred, Alloc> const&);
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        bool operator!=(unordered_set<Value, Hash, Pred, Alloc> const&,
            unordered_set<Value, Hash, Pred, Alloc> const&);
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        void swap(unordered_set<Value, Hash, Pred, Alloc>&,
            unordered_set<Value, Hash, Pred, Alloc>&);
    template<typename Value, typename Hash = boost::hash<Value>,
            typename Pred = std::equal_to<Value>,
            typename Alloc = std::allocator<Value> >
        class unordered_multiset;
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        bool operator==(unordered_multiset<Value, Hash, Pred, Alloc> const&,
            unordered_multiset<Value, Hash, Pred, Alloc> const&);
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        bool operator!=(unordered_multiset<Value, Hash, Pred, Alloc> const&,
            unordered_multiset<Value, Hash, Pred, Alloc> const&);
    template<typename Value, typename Hash, typename Pred, typename Alloc>
        void swap(unordered_multiset<Value, Hash, Pred, Alloc>&,
            unordered_multiset<Value, Hash, Pred, Alloc>&);
}
```

Class template unordered_set

boost::unordered_set — An unordered associative container that stores unique values.

Synopsis

```
// In header: <boost/unordered_set.hpp>

template<typename Value, typename Hash = boost::hash<Value>,
        typename Pred = std::equal_to<Value>,
        typename Alloc = std::allocator<Value> >
class unordered_set {
public:
    // types
    typedef Value key_type;
    typedef Value value_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/copy/destruct
    explicit unordered_set(size_type = implementation-defined,
                          hasher const& = hasher(),
                          key_equal const& = key_equal(),
                          allocator_type const& = allocator_type());
    template<typename InputIterator>
        unordered_set(InputIterator, InputIterator,
                      size_type = implementation-defined,
                      hasher const& = hasher(), key_equal const& = key_equal(),
                      allocator_type const& = allocator_type());
    unordered_set(unordered_set const&);
    unordered_set(unordered_set &&);
    explicit unordered_set(Allocator const&);
    unordered_set(unordered_set const&, Allocator const&);
    ~unordered_set();
    unordered_set& operator=(unordered_set const&);
    unordered_set& operator=(unordered_set &&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

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

```

// modifiers
template<typename... Args> std::pair<iterator, bool> emplace(Args&&...);
template<typename... Args> iterator emplace_hint(const_iterator, Args&&...);
std::pair<iterator, bool> insert(value_type const&);
iterator insert(const_iterator, value_type const&);
template<typename InputIterator> void insert(InputIterator, InputIterator);
iterator erase(const_iterator);
size_type erase(key_type const&);
iterator erase(const_iterator, const_iterator);
void quick_erase(const_iterator);
void erase_return_void(const_iterator);
void clear();
void swap(unordered_set&);

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

// lookup
iterator find(key_type const&);
const_iterator find(key_type const&) 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(key_type const&) const;
std::pair<iterator, iterator> equal_range(key_type const&);
std::pair<const_iterator, const_iterator> equal_range(key_type const&) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type) const;
size_type bucket(key_type const&) const;
local_iterator begin(size_type);
const_local_iterator begin(size_type) const;
local_iterator end(size_type);
const_local_iterator end(size_type) const;
const_local_iterator cbegin(size_type) const;
const_local_iterator cend(size_type);

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float);
void rehash(size_type);
};

// Equality Comparisons
template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator==(unordered_set<Value, Hash, Pred, Alloc> const&,
                   unordered_set<Value, Hash, Pred, Alloc> const&);
template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator!=(unordered_set<Value, Hash, Pred, Alloc> const&,

```

```

        unordered_set<Value, Hash, Pred, Alloc> const&);

// swap
template<typename Value, typename Hash, typename Pred, typename Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>&,
              unordered_set<Value, Hash, Pred, Alloc>&);

```

Description

Based on chapter 23 of [the working draft of the C++ standard \[n2960\]](#). But without the updated rules for allocators.

Template Parameters

<i>Value</i>	Value must be Assignable and CopyConstructible
<i>Hash</i>	A unary function object type that acts a hash function for a <code>Value</code> . It takes a single argument of type <code>Value</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>Value</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>Alloc</i>	An allocator whose value type is the same as the container's value type.

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`.

`unordered_set` public types

1. `typedef Value key_type;`
2. `typedef Value value_type;`
3. `typedef Hash hasher;`
4. `typedef Pred key_equal;`
5. `typedef Alloc allocator_type;`
6. `typedef typename allocator_type::pointer pointer;`
7. `typedef typename allocator_type::const_pointer const_pointer;`
8. `typedef typename allocator_type::reference reference;`
9. `typedef typename allocator_type::const_reference const_reference;`
10. `typedef implementation-defined size_type;`
An unsigned integral type.
`size_type` can represent any non-negative value of `difference_type`.
11. `typedef implementation-defined difference_type;`
A signed integral type.
Is identical to the difference type of `iterator` and `const_iterator`.

12. 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.

13. typedef *implementation-defined* const_iterator;

A constant iterator whose value type is value_type.

The iterator category is at least a forward iterator.

14. typedef *implementation-defined* local_iterator;

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.

15. typedef *implementation-defined* const_local_iterator;

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.

unordered_set public construct/copy/destruct

```
1. explicit unordered_set(size_type n = implementation-defined,
                        hasher const& hf = hasher(),
                        key_equal const& eq = key_equal(),
                        allocator_type const& a = allocator_type());
```

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

Postconditions: `size() == 0`

```
2. template<typename InputIterator>
    unordered_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  hasher const& hf = hasher(),
                  key_equal const& eq = key_equal(),
                  allocator_type const& a = allocator_type());
```

Constructs an empty container with at least n buckets, using hf as the hash function, eq 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.

```
3. unordered_set(unordered_set const&);
```

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

Requires: `value_type` is copy constructible

```
4. unordered_set(unordered_set &&);
```

The move constructor.

Notes: This is emulated on compilers without rvalue references.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

5. `explicit unordered_set(Allocator const& a);`

Constructs an empty container, using allocator a.

6. `unordered_set(unordered_set const& x, Allocator const& a);`

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

7. `~unordered_set();`

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

```
unordered_set& operator=(unordered_set const&);
```

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

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_set)` in order to emulate move semantics.

Requires: `value_type` is copy constructible

```
unordered_set& operator=(unordered_set &&);
```

The move assignment operator.

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_set)` in order to emulate move semantics.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

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

unordered_set size and capacity

1. `bool empty() const;`

Returns: `size() == 0`

2. `size_type size() const;`

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

3. `size_type max_size() const;`

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

unordered_set iterators

1. `iterator begin();`
`const_iterator begin() const;`

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.

```
2. iterator end();  
   const_iterator end() const;
```

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

```
3. const_iterator cbegin() const;
```

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

```
4. const_iterator cend() const;
```

Returns: A constant iterator which refers to the past-the-end value for the container.

unordered_set modifiers

```
1. template<typename... 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 value.

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

```
2. template<typename... Args>  
   iterator emplace_hint(const_iterator hint, 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 value.

`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 value.

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 value.

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

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

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

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.

4.

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

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

`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 value.

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 value.

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.

5.

```
template<typename 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 value.

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.

6.

```
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`.

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: When the number of elements is a lot smaller than the number of buckets this function can be very inefficient as it has to search through empty buckets for the next element, in order to return the iterator. The method `quick_erase` is faster, but has yet to be standardized.

7.

```
size_type erase(key_type const& 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`.

8.

```
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.

9.

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

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 is faster than `erase` as it doesn't have to find the next element in the container - a potentially costly operation.

As it hasn't been standardized, it's likely that this may change in the future.

10.

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

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 is now deprecated, use `quick_return` instead. Although be warned that as that isn't standardized yet, it could also change.

11.

```
void clear();
```

Erases all elements in the container.

Postconditions: `size() == 0`

Throws: Never throws an exception.

12.

```
void swap(unordered_set&);
```

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

unordered_set observers

1.

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

Returns: The container's hash function.

2.

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

Returns: The container's key equality predicate.

unordered_set lookup

```
1. iterator find(key_type const& k);
   const_iterator find(key_type const& 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;
```

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

Notes: The templated overloads are a non-standard extensions which allows 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.

```
2. size_type count(key_type const& k) const;
```

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

```
3. std::pair<iterator, iterator> equal_range(key_type const& k);
   std::pair<const_iterator, const_iterator> equal_range(key_type const& 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()).

unordered_set bucket interface

```
1. size_type bucket_count() const;
```

Returns: The number of buckets.

```
2. size_type max_bucket_count() const;
```

Returns: An upper bound on the number of buckets.

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

Requires: $n < \text{bucket_count}()$

Returns: The number of elements in bucket n.

```
4. size_type bucket(key_type const& 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()

```
5. 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.

```
6. 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 .

```
7. 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 .

```
8. const_local_iterator cend(size_type n);
```

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

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

unordered_set hash policy

```
1. float load_factor() const;
```

Returns: The average number of elements per bucket.

```
2. float max_load_factor() const;
```

Returns: Returns the current maximum load factor.

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

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

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

Changes the number of buckets so that there at least n buckets, and so that the load factor is less than the maximum load factor.

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.

unordered_set Equality Comparisons

```
1. template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator==(unordered_set<Value, Hash, Pred, Alloc> const& x,
                    unordered_set<Value, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

```
2. template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator!=(unordered_set<Value, Hash, Pred, Alloc> const& x,
                    unordered_set<Value, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

unordered_set swap

1.

```
template<typename Value, typename Hash, typename Pred, typename Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
          unordered_set<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of Hash or Pred.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

Class template unordered_multiset

boost::unordered_multiset — An unordered associative container that stores values. The same key can be stored multiple times.

Synopsis

```
// In header: <boost/unordered_set.hpp>

template<typename Value, typename Hash = boost::hash<Value>,
        typename Pred = std::equal_to<Value>,
        typename Alloc = std::allocator<Value> >
class unordered_multiset {
public:
    // types
    typedef Value key_type;
    typedef Value value_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/copy/destruct
    explicit unordered_multiset(size_type = implementation-defined,
                              hasher const& = hasher(),
                              key_equal const& = key_equal(),
                              allocator_type const& = allocator_type());

    template<typename InputIterator>
        unordered_multiset(InputIterator, InputIterator,
                            size_type = implementation-defined,
                            hasher const& = hasher(),
                            key_equal const& = key_equal(),
                            allocator_type const& = allocator_type());

    unordered_multiset(unordered_multiset const&);
    unordered_multiset(unordered_multiset &&);
    explicit unordered_multiset(Allocator const&);
    unordered_multiset(unordered_multiset const&, Allocator const&);
    ~unordered_multiset();
    unordered_multiset& operator=(unordered_multiset const&);
    unordered_multiset& operator=(unordered_multiset &&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

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

```

// modifiers
template<typename... Args> iterator emplace(Args&&...);
template<typename... Args> iterator emplace_hint(const_iterator, Args&&...);
iterator insert(value_type const&);
iterator insert(const_iterator, value_type const&);
template<typename InputIterator> void insert(InputIterator, InputIterator);
iterator erase(const_iterator);
size_type erase(key_type const&);
iterator erase(const_iterator, const_iterator);
void quick_erase(const_iterator);
void erase_return_void(const_iterator);
void clear();
void swap(unordered_multiset&);

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

// lookup
iterator find(key_type const&);
const_iterator find(key_type const&) 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(key_type const&) const;
std::pair<iterator, iterator> equal_range(key_type const&);
std::pair<const_iterator, const_iterator> equal_range(key_type const&) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type) const;
size_type bucket(key_type const&) const;
local_iterator begin(size_type);
const_local_iterator begin(size_type) const;
local_iterator end(size_type);
const_local_iterator end(size_type) const;
const_local_iterator cbegin(size_type) const;
const_local_iterator cend(size_type);

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float);
void rehash(size_type);
};

// Equality Comparisons
template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator==(unordered_multiset<Value, Hash, Pred, Alloc> const&,
                   unordered_multiset<Value, Hash, Pred, Alloc> const&);
template<typename Value, typename Hash, typename Pred, typename Alloc>
    bool operator!=(unordered_multiset<Value, Hash, Pred, Alloc> const&,

```

```

        unordered_multiset<Value, Hash, Pred, Alloc> const&);

// swap
template<typename Value, typename Hash, typename Pred, typename Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>&,
              unordered_multiset<Value, Hash, Pred, Alloc>&);

```

Description

Based on chapter 23 of [the working draft of the C++ standard \[n2960\]](#). But without the updated rules for allocators.

Template Parameters

<i>Value</i>	Value must be Assignable and CopyConstructible
<i>Hash</i>	A unary function object type that acts a hash function for a <code>Value</code> . It takes a single argument of type <code>Value</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>Value</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>Alloc</i>	An allocator whose value type is the same as the container's value type.

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`.

`unordered_multiset` public types

1. `typedef Value key_type;`
2. `typedef Value value_type;`
3. `typedef Hash hasher;`
4. `typedef Pred key_equal;`
5. `typedef Alloc allocator_type;`
6. `typedef typename allocator_type::pointer pointer;`
7. `typedef typename allocator_type::const_pointer const_pointer;`
8. `typedef typename allocator_type::reference reference;`
9. `typedef typename allocator_type::const_reference const_reference;`
10. `typedef implementation-defined size_type;`

An unsigned integral type.

`size_type` can represent any non-negative value of `difference_type`.

11. `typedef implementation-defined difference_type;`

A signed integral type.

Is identical to the difference type of `iterator` and `const_iterator`.

12. 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.

13. typedef *implementation-defined* const_iterator;

A constant iterator whose value type is value_type.

The iterator category is at least a forward iterator.

14. typedef *implementation-defined* local_iterator;

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.

15. typedef *implementation-defined* const_local_iterator;

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.

unordered_multiset public construct/copy/destruct

```
1. explicit unordered_multiset(size_type n = implementation-defined,
                             hasher const& hf = hasher(),
                             key_equal const& eq = key_equal(),
                             allocator_type const& a = allocator_type());
```

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

Postconditions: `size() == 0`

```
2. template<typename InputIterator>
   unordered_multiset(InputIterator f, InputIterator l,
                     size_type n = implementation-defined,
                     hasher const& hf = hasher(),
                     key_equal const& eq = key_equal(),
                     allocator_type const& a = allocator_type());
```

Constructs an empty container with at least n buckets, using hf as the hash function, eq 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.

```
3. unordered_multiset(unordered_multiset const&);
```

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

Requires: `value_type` is copy constructible

```
4. unordered_multiset(unordered_multiset &&);
```

The move constructor.

Notes: This is emulated on compilers without rvalue references.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

5. `explicit unordered_multiset(Allocator const& a);`

Constructs an empty container, using allocator a.

6. `unordered_multiset(unordered_multiset const& x, Allocator const& a);`

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

7. `~unordered_multiset();`

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

```
unordered_multiset& operator=(unordered_multiset const&);
```

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

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_multiset)` in order to emulate move semantics.

Requires: `value_type` is copy constructible

```
unordered_multiset& operator=(unordered_multiset &&);
```

The move assignment operator.

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_multiset)` in order to emulate move semantics.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

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

unordered_multiset size and capacity

1. `bool empty() const;`

Returns: `size() == 0`

2. `size_type size() const;`

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

3. `size_type max_size() const;`

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

unordered_multiset iterators

1. `iterator begin();`
`const_iterator begin() const;`

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.

```
2. iterator end();
   const_iterator end() const;
```

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

```
3. const_iterator cbegin() const;
```

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

```
4. const_iterator cend() const;
```

Returns: A constant iterator which refers to the past-the-end value for the container.

unordered_multiset modifiers

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

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

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

```
2. template<typename... Args>
   iterator emplace_hint(const_iterator hint, Args&&... args);
```

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

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

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 value.

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

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

Inserts `obj` in the container.

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.

4.

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

Inserts obj in the container.

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

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 value.

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.

5.

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

Inserts a range of elements 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.

6.

```
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 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: When the number of elements is a lot smaller than the number of buckets this function can be very inefficient as it has to search through empty buckets for the next element, in order to return the iterator. The method `quick_erase` is faster, but has yet to be standardized.

7.

```
size_type erase(key_type const& 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`.

8.

```
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.

9.

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

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 is faster than [erase](#) as it doesn't have to find the next element in the container - a potentially costly operation.

As it hasn't been standardized, it's likely that this may change in the future.

10.

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

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 is now deprecated, use `quick_return` instead. Although be warned that as that isn't standardized yet, it could also change.

11.

```
void clear();
```

Erases all elements in the container.

Postconditions: `size() == 0`

Throws: Never throws an exception.

12.

```
void swap(unordered_multiset&);
```

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

`unordered_multiset` observers

1.

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

Returns: The container's hash function.

2.

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

Returns: The container's key equality predicate.

unordered_multiset lookup

```
1. iterator find(key_type const& k);
   const_iterator find(key_type const& 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;
```

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

Notes: The templated overloads are a non-standard extensions which allows 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.

```
2. size_type count(key_type const& k) const;
```

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

```
3. std::pair<iterator, iterator> equal_range(key_type const& k);
   std::pair<const_iterator, const_iterator> equal_range(key_type const& 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()).

unordered_multiset bucket interface

```
1. size_type bucket_count() const;
```

Returns: The number of buckets.

```
2. size_type max_bucket_count() const;
```

Returns: An upper bound on the number of buckets.

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

Requires: $n < \text{bucket_count}()$

Returns: The number of elements in bucket n.

```
4. size_type bucket(key_type const& 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()

```
5. 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.

6.

```
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 .

7.

```
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 .

8.

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

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

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

unordered_multiset hash policy

1.

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

Returns: The average number of elements per bucket.

2.

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

Returns: Returns the current maximum load factor.

3.

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

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

4.

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

Changes the number of buckets so that there at least n buckets, and so that the load factor is less than the maximum load factor.

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.

unordered_multiset Equality Comparisons

1.

```
template<typename Value, typename Hash, typename Pred, typename Alloc>  
bool operator==(unordered_multiset<Value, Hash, Pred, Alloc> const& x,  
                unordered_multiset<Value, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

2.

```
template<typename Value, typename Hash, typename Pred, typename Alloc>  
bool operator!=(unordered_multiset<Value, Hash, Pred, Alloc> const& x,  
                unordered_multiset<Value, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

unordered_multiset swap

1.

```
template<typename Value, typename Hash, typename Pred, typename Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
         unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of Hash or Pred.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

Header **<boost/unordered_map.hpp>**

```
namespace boost {
    template<typename Key, typename Mapped, typename Hash = boost::hash<Key>,
            typename Pred = std::equal_to<Key>,
            typename Alloc = std::allocator<std::pair<Key const, Mapped> > >
        class unordered_map;
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        bool operator==(unordered_map<Key, Mapped, Hash, Pred, Alloc> const&,
                        unordered_map<Key, Mapped, Hash, Pred, Alloc> const&);
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        bool operator!=(unordered_map<Key, Mapped, Hash, Pred, Alloc> const&,
                        unordered_map<Key, Mapped, Hash, Pred, Alloc> const&);
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        void swap(unordered_map<Key, Mapped, Hash, Pred, Alloc>&,
                  unordered_map<Key, Mapped, Hash, Pred, Alloc>&);
    template<typename Key, typename Mapped, typename Hash = boost::hash<Key>,
            typename Pred = std::equal_to<Key>,
            typename Alloc = std::allocator<std::pair<Key const, Mapped> > >
        class unordered_multimap;
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        bool operator==(unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&,
                        unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&);
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        bool operator!=(unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&,
                        unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&);
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
        void swap(unordered_multimap<Key, Mapped, Hash, Pred, Alloc>&,
                  unordered_multimap<Key, Mapped, Hash, Pred, Alloc>&);
}
```

Class template unordered_map

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

Synopsis

```
// In header: <boost/unordered_map.hpp>

template<typename Key, typename Mapped, typename Hash = boost::hash<Key>,
        typename Pred = std::equal_to<Key>,
        typename Alloc = std::allocator<std::pair<Key const, Mapped> > >
class unordered_map {
public:
    // types
    typedef Key key_type;
    typedef std::pair<Key const, Mapped> value_type;
    typedef Mapped mapped_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/copy/destruct
    explicit unordered_map(size_type = implementation-defined,
                          hasher const& = hasher(),
                          key_equal const& = key_equal(),
                          allocator_type const& = allocator_type());
    template<typename InputIterator>
    unordered_map(InputIterator, InputIterator,
                  size_type = implementation-defined,
                  hasher const& = hasher(), key_equal const& = key_equal(),
                  allocator_type const& = allocator_type());
    unordered_map(unordered_map const&);
    unordered_map(unordered_map &&);
    explicit unordered_map(Allocator const&);
    unordered_map(unordered_map const&, Allocator const&);
    ~unordered_map();
    unordered_map& operator=(unordered_map const&);
    unordered_map& operator=(unordered_map &&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

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

```

// modifiers
template<typename... Args> std::pair<iterator, bool> emplace(Args&&...);
template<typename... Args> iterator emplace_hint(const_iterator, Args&&...);
std::pair<iterator, bool> insert(value_type const&);
iterator insert(const_iterator, value_type const&);
template<typename InputIterator> void insert(InputIterator, InputIterator);
iterator erase(const_iterator);
size_type erase(key_type const&);
iterator erase(const_iterator, const_iterator);
void quick_erase(const_iterator);
void erase_return_void(const_iterator);
void clear();
void swap(unordered_map&);

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

// lookup
iterator find(key_type const&);
const_iterator find(key_type const&) 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(key_type const&) const;
std::pair<iterator, iterator> equal_range(key_type const&);
std::pair<const_iterator, const_iterator> equal_range(key_type const&) const;
mapped_type& operator[](key_type const&);
Mapped& at(key_type const&);
Mapped const& at(key_type const&) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type) const;
size_type bucket(key_type const&) const;
local_iterator begin(size_type);
const_local_iterator begin(size_type) const;
local_iterator end(size_type);
const_local_iterator end(size_type) const;
const_local_iterator cbegin(size_type) const;
const_local_iterator cend(size_type);

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float);
void rehash(size_type);
};

// Equality Comparisons
template<typename Key, typename Mapped, typename Hash, typename Pred,
         typename Alloc>
    bool operator==(unordered_map<Key, Mapped, Hash, Pred, Alloc> const&,
                    unordered_map<Key, Mapped, Hash, Pred, Alloc> const&);
template<typename Key, typename Mapped, typename Hash, typename Pred,

```

```

        typename Alloc>
    bool operator!=(unordered_map<Key, Mapped, Hash, Pred, Alloc> const&,
                    unordered_map<Key, Mapped, Hash, Pred, Alloc> const&);

    // swap
    template<typename Key, typename Mapped, typename Hash, typename Pred,
            typename Alloc>
    void swap(unordered_map<Key, Mapped, Hash, Pred, Alloc>&,
              unordered_map<Key, Mapped, Hash, Pred, Alloc>&);

```

Description

Based on chapter 23 of [the working draft of the C++ standard \[n2960\]](#). But without the updated rules for allocators.

Template Parameters

<i>Key</i>	Key must be Assignable and CopyConstructible.
<i>Mapped</i>	Mapped must be CopyConstructible
<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>Alloc</i>	An allocator whose value type is the same as the container's value type.

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`.

`unordered_map` public types

1. `typedef Key key_type;`
2. `typedef std::pair<Key const, Mapped> value_type;`
3. `typedef Mapped mapped_type;`
4. `typedef Hash hasher;`
5. `typedef Pred key_equal;`
6. `typedef Alloc allocator_type;`
7. `typedef typename allocator_type::pointer pointer;`
8. `typedef typename allocator_type::const_pointer const_pointer;`
9. `typedef typename allocator_type::reference reference;`
10. `typedef typename allocator_type::const_reference const_reference;`
11. `typedef implementation-defined size_type;`

An unsigned integral type.

`size_type` can represent any non-negative value of `difference_type`.

12. typedef *implementation-defined* difference_type;

A signed integral type.

Is identical to the difference type of iterator and const_iterator.

13. typedef *implementation-defined* iterator;

A iterator whose value type is value_type.

The iterator category is at least a forward iterator.

Convertible to const_iterator.

14. typedef *implementation-defined* const_iterator;

A constant iterator whose value type is value_type.

The iterator category is at least a forward iterator.

15. typedef *implementation-defined* local_iterator;

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.

16. typedef *implementation-defined* const_local_iterator;

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.

unordered_map public construct/copy/destruct

```
1. explicit unordered_map(size_type n = implementation-defined,
    hasher const& hf = hasher(),
    key_equal const& eq = key_equal(),
    allocator_type const& a = allocator_type());
```

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

Postconditions: `size() == 0`

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

Constructs an empty container with at least n buckets, using hf as the hash function, eq 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.

```
3. unordered_map(unordered_map const&);
```

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

Requires: `value_type` is copy constructible

```
4. unordered_map(unordered_map &&);
```

The move constructor.

Notes: This is emulated on compilers without rvalue references.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

5.

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

Constructs an empty container, using allocator `a`.

6.

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

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

7.

```
~unordered_map();
```

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

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

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

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_map)` in order to emulate move semantics.

Requires: `value_type` is copy constructible

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

The move assignment operator.

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_map)` in order to emulate move semantics.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

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

unordered_map size and capacity

1.

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

Returns: `size() == 0`

2.

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

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

3.

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

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

unordered_map iterators

1.

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


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.

```
2. iterator end();  
   const_iterator end() const;
```

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

```
3. const_iterator cbegin() const;
```

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

```
4. const_iterator cend() const;
```

Returns: A constant iterator which refers to the past-the-end value for the container.

unordered_map modifiers

```
1. template<typename... 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.

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

```
2. template<typename... Args>  
   iterator emplace_hint(const_iterator hint, 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.

`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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

3.

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

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

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.

4.

```
iterator insert(const_iterator hint, value_type const& 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.

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.

5.

```
template<typename 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.

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.

6.

```
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`.

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: When the number of elements is a lot smaller than the number of buckets this function can be very inefficient as it has to search through empty buckets for the next element, in order to return the iterator. The method [quick_erase](#) is faster, but has yet to be standardized.

7.

```
size_type erase(key_type const& 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.

8.

```
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.

9.

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

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 is faster than [erase](#) as it doesn't have to find the next element in the container - a potentially costly operation.

As it hasn't been standardized, it's likely that this may change in the future.

10.

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

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 is now deprecated, use `quick_return` instead. Although be warned that as that isn't standardized yet, it could also change.

11.

```
void clear();
```

Erases all elements in the container.

Postconditions: `size() == 0`

Throws: Never throws an exception.

12.

```
void swap(unordered_map&);
```

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of key_equal or hasher.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

unordered_map observers

1.

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

Returns: The container's hash function.

2.

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

Returns: The container's key equality predicate.

unordered_map lookup

```
1. iterator find(key_type const& k);
   const_iterator find(key_type const& 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;
```

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

Notes: The templated overloads are a non-standard extensions which allows 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.

```
2. size_type count(key_type const& k) const;
```

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

```
3. std::pair<iterator, iterator> equal_range(key_type const& k);
   std::pair<const_iterator, const_iterator> equal_range(key_type const& 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()).

```
4. mapped_type& operator[] (key_type const& 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 factor to be greater to or equal to the maximum load factor.

Pointers and references to elements are never invalidated.

```
5. Mapped& at(key_type const& k);
   Mapped const& at(key_type const& 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: This is not specified in the draft standard, but that is probably an oversight. The issue has been raised in <http://ericniebler.com/2014/05/24/unordered-map-at/>.

unordered_map bucket interface

```
1. size_type bucket_count() const;
```

Returns: The number of buckets.

2. `size_type max_bucket_count() const;`

Returns: An upper bound on the number of buckets.

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

Requires: $n < \text{bucket_count}()$

Returns: The number of elements in bucket n .

4. `size_type bucket(key_type const& 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()`

5. `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 .

6. `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 .

7. `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 .

8. `const_local_iterator cend(size_type n);`

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

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

unordered_map hash policy

1. `float load_factor() const;`

Returns: The average number of elements per bucket.

2. `float max_load_factor() const;`

Returns: Returns the current maximum load factor.

3. `void max_load_factor(float z);`

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

4. `void rehash(size_type n);`

Changes the number of buckets so that there at least n buckets, and so that the load factor is less than the maximum load factor.

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.

`unordered_map` Equality Comparisons

```
1. template<typename Key, typename Mapped, typename Hash, typename Pred,
           typename Alloc>
    bool operator==(unordered_map<Key, Mapped, Hash, Pred, Alloc> const& x,
                    unordered_map<Key, Mapped, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

```
2. template<typename Key, typename Mapped, typename Hash, typename Pred,
           typename Alloc>
    bool operator!=(unordered_map<Key, Mapped, Hash, Pred, Alloc> const& x,
                    unordered_map<Key, Mapped, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

`unordered_map` swap

```
1. template<typename Key, typename Mapped, typename Hash, typename Pred,
           typename Alloc>
    void swap(unordered_map<Key, Mapped, Hash, Pred, Alloc>& x,
              unordered_map<Key, Mapped, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of Hash or Pred.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

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

```
// In header: <boost/unordered_map.hpp>

template<typename Key, typename Mapped, typename Hash = boost::hash<Key>,
        typename Pred = std::equal_to<Key>,
        typename Alloc = std::allocator<std::pair<Key const, Mapped> > >
class unordered_multimap {
public:
    // types
    typedef Key key_type;
    typedef std::pair<Key const, Mapped> value_type;
    typedef Mapped mapped_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/copy/destroy
    explicit unordered_multimap(size_type = implementation-defined,
                              hasher const& = hasher(),
                              key_equal const& = key_equal(),
                              allocator_type const& = allocator_type());

    template<typename InputIterator>
    unordered_multimap(InputIterator, InputIterator,
                      size_type = implementation-defined,
                      hasher const& = hasher(),
                      key_equal const& = key_equal(),
                      allocator_type const& = allocator_type());

    unordered_multimap(unordered_multimap const&);
    unordered_multimap(unordered_multimap &&);
    explicit unordered_multimap(Allocator const&);
    unordered_multimap(unordered_multimap const&, Allocator const&);
    ~unordered_multimap();
    unordered_multimap& operator=(unordered_multimap const&);
    unordered_multimap& operator=(unordered_multimap &&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    // iterators
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
```

```

const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template<typename... Args> iterator emplace(Args&&...);
template<typename... Args> iterator emplace_hint(const_iterator, Args&&...);
iterator insert(value_type const&);
iterator insert(const_iterator, value_type const&);
template<typename InputIterator> void insert(InputIterator, InputIterator);
iterator erase(const_iterator);
size_type erase(key_type const&);
iterator erase(const_iterator, const_iterator);
void quick_erase(const_iterator);
void erase_return_void(const_iterator);
void clear();
void swap(unordered_multimap&);

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

// lookup
iterator find(key_type const&);
const_iterator find(key_type const&) 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(key_type const&) const;
std::pair<iterator, iterator> equal_range(key_type const&);
std::pair<const_iterator, const_iterator> equal_range(key_type const&) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type) const;
size_type bucket(key_type const&) const;
local_iterator begin(size_type);
const_local_iterator begin(size_type) const;
local_iterator end(size_type);
const_local_iterator end(size_type) const;
const_local_iterator cbegin(size_type) const;
const_local_iterator cend(size_type);

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float);
void rehash(size_type);
};

// Equality Comparisons
template<typename Key, typename Mapped, typename Hash, typename Pred,
         typename Alloc>
    bool operator==(unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&,
                    unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&);
template<typename Key, typename Mapped, typename Hash, typename Pred,
         typename Alloc>

```



```

bool operator!=(unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&,
                unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const&);

// swap
template<typename Key, typename Mapped, typename Hash, typename Pred,
        typename Alloc>
void swap(unordered_multimap<Key, Mapped, Hash, Pred, Alloc>&,
          unordered_multimap<Key, Mapped, Hash, Pred, Alloc>&);

```

Description

Based on chapter 23 of [the working draft of the C++ standard \[n2960\]](#). But without the updated rules for allocators.

Template Parameters

<i>Key</i>	Key must be Assignable and CopyConstructible.
<i>Mapped</i>	Mapped must be CopyConstructible
<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>Alloc</i>	An allocator whose value type is the same as the container's value type.

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`.

`unordered_multimap` public types

1. `typedef Key key_type;`
2. `typedef std::pair<Key const, Mapped> value_type;`
3. `typedef Mapped mapped_type;`
4. `typedef Hash hasher;`
5. `typedef Pred key_equal;`
6. `typedef Alloc allocator_type;`
7. `typedef typename allocator_type::pointer pointer;`
8. `typedef typename allocator_type::const_pointer const_pointer;`
9. `typedef typename allocator_type::reference reference;`
10. `typedef typename allocator_type::const_reference const_reference;`
11. `typedef implementation-defined size_type;`

An unsigned integral type.

`size_type` can represent any non-negative value of `difference_type`.

12. typedef *implementation-defined* difference_type;

A signed integral type.

Is identical to the difference type of iterator and const_iterator.

13. typedef *implementation-defined* iterator;

A iterator whose value type is value_type.

The iterator category is at least a forward iterator.

Convertible to const_iterator.

14. typedef *implementation-defined* const_iterator;

A constant iterator whose value type is value_type.

The iterator category is at least a forward iterator.

15. typedef *implementation-defined* local_iterator;

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.

16. typedef *implementation-defined* const_local_iterator;

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.

unordered_multimap public construct/copy/destruct

```
1. explicit unordered_multimap(size_type n = implementation-defined,
                             hasher const& hf = hasher(),
                             key_equal const& eq = key_equal(),
                             allocator_type const& a = allocator_type());
```

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

Postconditions: `size() == 0`

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

Constructs an empty container with at least n buckets, using hf as the hash function, eq 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.

```
3. unordered_multimap(unordered_multimap const&);
```

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

Requires: `value_type` is copy constructible

```
4. unordered_multimap(unordered_multimap &&);
```

The move constructor.

Notes: This is emulated on compilers without rvalue references.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

5.

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

Constructs an empty container, using allocator `a`.

6.

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

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

7.

```
~unordered_multimap();
```

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

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

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

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_multimap)` in order to emulate move semantics.

Requires: `value_type` is copy constructible

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

The move assignment operator.

Notes: On compilers without rvalue references, there is a single assignment operator with the signature `operator=(unordered_multimap)` in order to emulate move semantics.

Requires: `value_type` is move constructible. (TODO: This is not actually required in this implementation).

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

unordered_multimap size and capacity

1.

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

Returns: `size() == 0`

2.

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

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

3.

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

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

unordered_multimap iterators

1.

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

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.

2.

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

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

3.

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

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

4.

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

Returns: A constant iterator which refers to the past-the-end value for the container.

unordered_multimap modifiers

1.

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

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

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

2.

```
template<typename... Args>  
iterator emplace_hint(const_iterator hint, Args&&... args);
```

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

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

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.

If the compiler doesn't support variadic template arguments or rvalue references, this is emulated for up to 10 arguments, with no support for rvalue references or move semantics.

3.

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

Inserts `obj` in the container.

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.

4.

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

Inserts obj in the container.

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

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.

5.

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

Inserts a range of elements 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.

6.

```
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`.

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: When the number of elements is a lot smaller than the number of buckets this function can be very inefficient as it has to search through empty buckets for the next element, in order to return the iterator. The method [quick_erase](#) is faster, but has yet to be standardized.

7.

```
size_type erase(key_type const& 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`.

8.

```
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.

9.

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

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 is faster than `erase` as it doesn't have to find the next element in the container - a potentially costly operation.

As it hasn't been standardized, it's likely that this may change in the future.

10.

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

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 is now deprecated, use `quick_return` instead. Although be warned that as that isn't standardized yet, it could also change.

11.

```
void clear();
```

Erases all elements in the container.

Postconditions: `size() == 0`

Throws: Never throws an exception.

12.

```
void swap(unordered_multimap&);
```

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of `key_equal` or `hasher`.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

`unordered_multimap` observers

1.

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

Returns: The container's hash function.

2.

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

Returns: The container's key equality predicate.

unordered_multimap lookup

```

1. iterator find(key_type const& k);
   const_iterator find(key_type const& 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;

```

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

Notes: The templated overloads are a non-standard extensions which allows 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.

```

2. size_type count(key_type const& k) const;

```

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

```

3. std::pair<iterator, iterator> equal_range(key_type const& k);
   std::pair<const_iterator, const_iterator> equal_range(key_type const& 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())`.

unordered_multimap bucket interface

```

1. size_type bucket_count() const;

```

Returns: The number of buckets.

```

2. size_type max_bucket_count() const;

```

Returns: An upper bound on the number of buckets.

```

3. size_type bucket_size(size_type n) const;

```

Requires: `n < bucket_count()`

Returns: The number of elements in bucket `n`.

```

4. size_type bucket(key_type const& 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()`

```

5. 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`.

6.

```
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`.

7.

```
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`.

8.

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

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`.

unordered_multimap hash policy

1.

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

Returns: The average number of elements per bucket.

2.

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

Returns: Returns the current maximum load factor.

3.

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

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

4.

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

Changes the number of buckets so that there at least `n` buckets, and so that the load factor is less than the maximum load factor.

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.

unordered_multimap Equality Comparisons

1.

```
template<typename Key, typename Mapped, typename Hash, typename Pred,  
         typename Alloc>  
bool operator==(unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const& x,  
               unordered_multimap<Key, Mapped, Hash, Pred, Alloc> const& y);
```

Notes: This is a boost extension.

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

2.

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


Notes: This is a boost extension.

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

unordered_multimap swap

```
1. template<typename Key, typename Mapped, typename Hash, typename Pred,
           typename Alloc>
    void swap(unordered_multimap<Key, Mapped, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, Mapped, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`

Throws: If the allocators are equal, doesn't throw an exception unless it is thrown by the copy constructor or copy assignment operator of Hash or Pred.

Notes: For a discussion of the behavior when allocators aren't equal see [the implementation details](#).

Bibliography

Bibliography

C/C++ Users Journal. February, 2006. Pete Becker. "[STL and TR1: Part III - Unordered containers](#)".

An introduction to the standard unordered containers.