
7.9. Unordered Containers

The hash table, one important data structure for collections, was not part of the first version of the C++ standard library. They were not part of the original STL and the committee decided that the proposal for their inclusion in C++98 came too late. (At some point you have to stop introducing features and focus on the details. Otherwise, you never finish the work.) However, with TR1, containers with the characteristics of hash tables finally came into the standard.

Nevertheless, even before TR1, several implementations of hash tables were available in the C++ community. Libraries typically provided four kinds of hash tables: `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`. However, those hash tables have been implemented slightly differently. With TR1, a consolidated group of hash table-based containers was introduced. The features provided for the standardized classes combined existing implementations and didn't match any of them completely. To avoid name clashes, therefore, different class names were chosen. The decision was to provide all the existing associative containers with the prefix `unordered_`. This also demonstrates the most important difference between ordinary and the new associative containers: With the hash table-based implementations, the elements have no defined order. See [*N1456:HashTable*] for details about the design decisions for all the unordered containers.

Strictly speaking, the C++ standard library calls unordered containers "unordered associative containers." However, I will just use "unordered containers" when I refer to them. With "associative containers," I still refer to the "old" associative containers, which are provided since C++98 and implemented as binary trees (set, multiset, map, and multimap).

Conceptually, unordered containers contain all the elements you insert in an arbitrary order (see Figure 7.16). That is, you can consider the container to be a bag: you can put in elements, but when you open the bag to do something with all the elements, you access them in a random order. So, in contrast with (multi)sets and (multi)maps, there is no sorting criterion; in contrast with sequence containers, you have no semantics to put an element into a specific position.

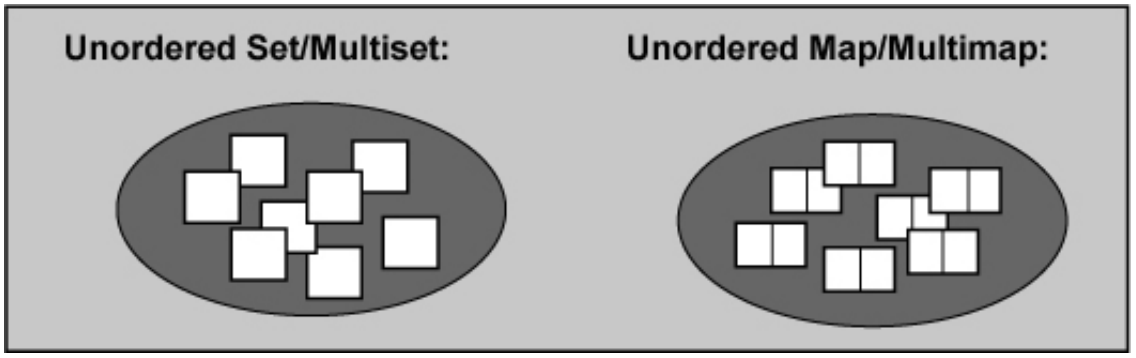

Figure 7.16. Unordered Containers

As with associative containers, the individual classes differ as follows:

- Unordered sets and multisets store single values of a specific type, whereas in unordered maps and multimaps, the elements are key/value pairs, where the key is used to store and find a specific element, including its associated value.
- Unordered sets and maps allow no duplicates, whereas unordered multisets and multimaps do.

To use an unordered set or multiset, you must include the header file `<unordered_set>`. To use an unordered map or multimap, you must include the header file `<unordered_map>`:

```
#include <unordered_set>
#include <unordered_map>
```

There, the types are defined as class templates inside namespace `std`:

Click here to view code image

```
namespace std {
    template <typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<T> >
             class unordered_set;

    template <typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<T> >
             class unordered_multiset;

    template <typename Key, typename T,
              typename Hash = hash<Key>,
```

```
                typename EqPred = equal_to<Key>,
                typename Allocator = allocator<pair<const Key, T> > >
              class unordered_map;

      template <typename Key, typename T,
                typename Hash = hash<Key>,
                typename EqPred = equal_to<Key>,
                typename Allocator = allocator<pair<const Key, T> > >
              class unordered_multimap;
  }
```

The elements of an unordered set or multiset may have any type `T` that is comparable.

For unordered maps and multimaps, the first template parameter is the type of the element's key, and the second template parameter is the type of the element's associated value. The elements of an unordered map or an unordered multimap may have any types `Key` and `T` that meet the following two requirements:

1. Both key and value must be copyable or movable.
2. The key must be comparable with the equivalence criterion.

Note that the element type (`value_type`) is a `pair<const` *Key* `,` *T* `>` .

The optional second/third template parameter defines the hash function. If a special hash function is not passed, the default hash function `hash<>` is used, which is provided as a function object in `<functional>` for all integral types, all floating-point types, pointers, strings, and some special types.[14] For all other value types, you must pass your own hash function, which is explained in Section 7.9.2, page 363, and Section 7.9.7, page 377.

[14] `error_code` , `thread::id` , `bitset<>` , and `vector<bool>`

The optional third/fourth template parameter defines an equivalence criterion: a predicate that is used to find elements. It should return whether two values are equal. If a special compare criterion is not passed, the default criterion `equal_to<>` is used, which compares the elements by comparing them with operator `==` (see Section 10.2.1, page 487, for details about `equal_to<>`).

The optional fourth/fifth template parameter defines the memory model (see Chapter 19). The default memory model is the model `allocator` , which is provided by the C++ standard library.

7.9.1. Abilities of Unordered Containers

All standardized unordered container classes are implemented as hash tables, which nonetheless still have a variety of implementation options. As usual, the C++ standard library does not specify all these implementation details to allow a variety of possible implementation options, but a few of the specified abilities of unordered containers are based on the following assumptions (see [*N1456:HashTable*]):

- The hash tables use the "chaining" approach, whereby a hash code is associated with a linked list. (This technique, also called "open hashing" or "closed addressing," should not be confused with "open addressing" or "closed hashing.")
- Whether these linked lists are singly or doubly linked is open to the implementers. For this reason, the standard guarantees only that the iterators are "at least" forward iterators.
- Various implementation strategies are possible for rehashing:
  - With the traditional approach, a complete reorganization of the internal data happens from time to time as a result of a single insert or erase operation.
  - With incremental hashing, a resizing of the number of bucket or slots is performed gradually, which is especially useful in real-time environments, where the price of enlarging a hash table all at once can be too high.

  Unordered containers allow both strategies and give no guarantee that conflicts with either of them.

Figure 7.17 shows the typical internal layout of an unordered set or multiset according to the minimal guarantees given by the C++ standard library. For each value to store, the hash function maps it to a bucket (slot) in the hash table. Each bucket manages a singly linked list containing all the elements for which the hash function yields the same value.

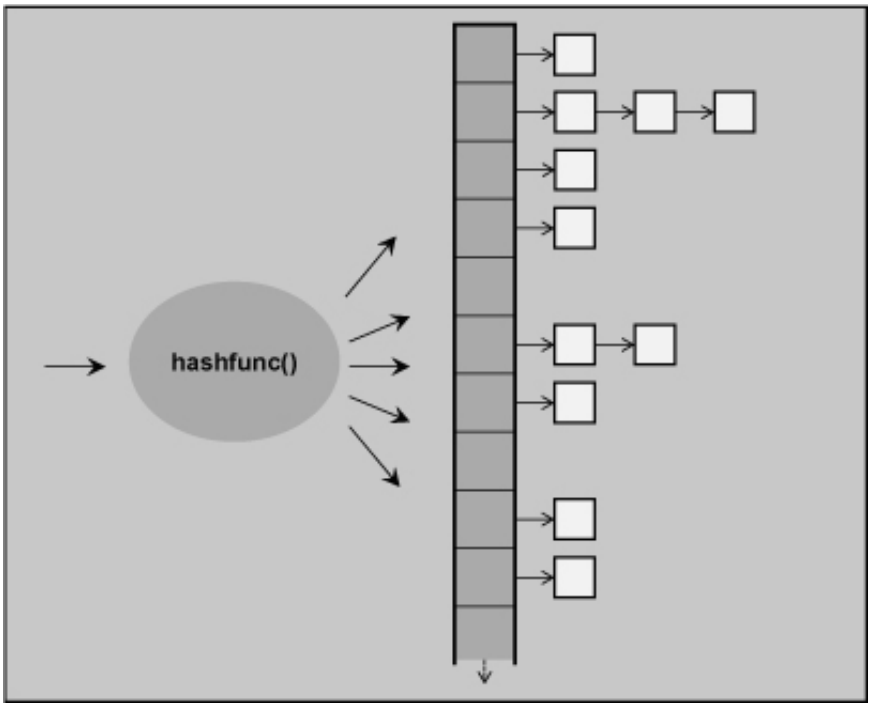

Figure 7.17. Internal Structure of Unordered Sets and Multisets

Figure 7.18 shows the typical internal layout of an unordered map or multimap according to the minimal guarantees given by the C++ standard library. For each element to store, which is a key/value pair, the hash function maps the value of the key to a bucket (slot) in the hash table. Each bucket manages a singly linked list containing all the elements for which the hash function yields the same value.

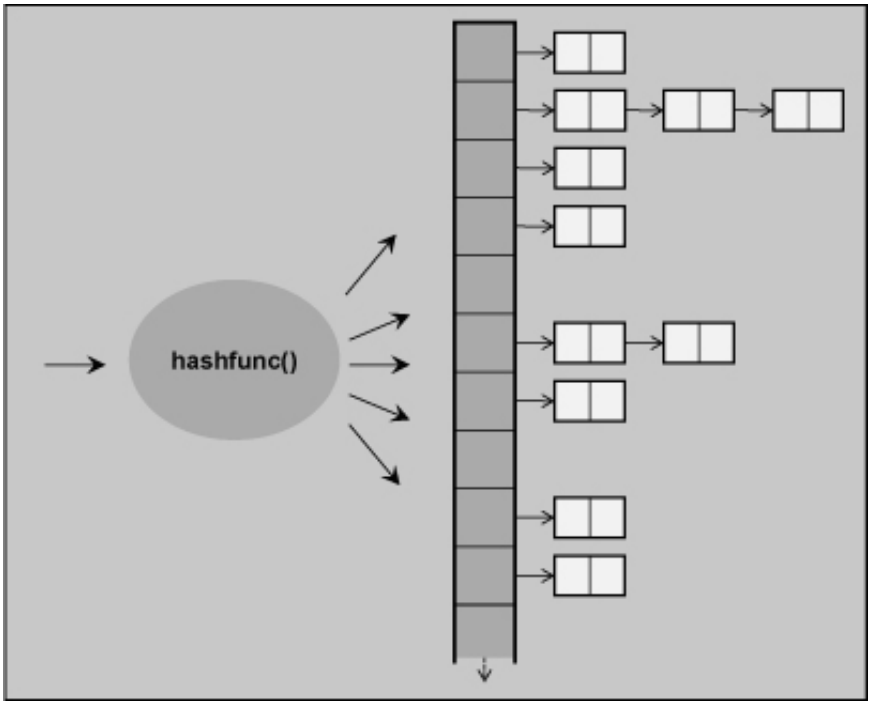

Figure 7.18. Internal Structure of Unordered Maps and Multimaps

The major advantage of using a hash table internally is its incredible running-time behavior. Assuming that the hashing strategy is well chosen and well implemented, you can guarantee amortized constant time for insertions, deletions, and element search ("amortized" because of the occasional rehashings, which can be large operations with linear complexity).

The expected behavior of nearly all the operations on unordered containers, including copy construction and assignment, element insertion and lookup, and equivalence comparison, depends on the quality of the hash function. If the hash function generates equal values for different elements, which also happens if an unordered container that allows duplicates is populated with equivalent values or keys, any hash table operation results in poor runtime performance. This is a fault not so much of the data structure itself but rather of its use by unenlightened clients.

Unordered containers also have some disadvantages over ordinary associative containers:

- Unordered containers don't provide operators `<`, `>`, `<=`, and `>=` to order multiple instances of these containers. However, `==` and `!=` are provided (since C++11).
- `lower_bound()` and `upper_bound()` are not provided.
- Because the iterators are guaranteed only to be forward iterators, reverse iterators, including `rbegin()`, `rend()`,

`crbegin()`, and `crend()`, are not supported, and you can't use algorithms that require bidirectional iterators, or at least this is not portable.

Because the (key) value of an element specifies its position — in this case, its bucket entry — you are *not* allowed to modify the (key) value of an element directly. Therefore, much as with associative containers, to modify the value of an element, you must remove the element that has the old value and insert a new element that has the new value. The interface reflects this behavior:

- Unordered containers don't provide operations for direct element access.
- Indirect access via iterators has the constraint that, from the iterator's point of view, the element's (key) value is constant.

As a programmer, you can specify parameters that influence the behavior of the hash table:

- You can specify the minimum number of buckets.
- You can (and sometimes have to) provide your own hash function.
- You can (and sometimes have to) provide your own equivalence criterion: a predicate that is used to find the right element among all entries in the bucket lists.
- You can specify a maximum load factor, which leads to automatic rehashing when the number of elements in the container grows.
- You can force rehashing.

But you can't influence the following behavior:

- The growth factor, which is the factor automatic rehashing uses to grow or shrink the list of buckets
- The minimum load factor, which is used to force rehashing when the number of elements in the container shrinks

Note that rehashing is possible only after a call to `insert()`, `rehash()`, `reserve()`, or `clear()`. This is a consequence of the guarantee that `erase()` never invalidates iterators, references, and pointers to other elements. Thus, if you delete hundreds of elements, the bucket size will not change. But if you insert one element afterward, the bucket size might shrink.

Also note that in containers that support equivalent keys — unordered multisets and multimaps — elements with equivalent keys are adjacent to each other when iterating over the elements of the container. Rehashing and other operations that internally change the order of elements preserve the relative order of elements with equivalent keys.

7.9.2. Creating and Controlling Unordered Containers

Hash tables are pretty complicated data structures. For this reason, you have a lot of abilities to define or query their behavior.

Create, Copy, and Destroy

Table 7.47 lists the constructors and destructors of unordered associative containers. Table 7.48 lists the types ***Unord*** that can be used with these constructors and destructors.

Table 7.47. Constructors and Destructors of Unordered Containers

Operation	Effect
Unord `c`	Default constructor; creates an empty unordered container without any elements
Unord `c(`*bnum*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets
Unord `c(`*bnum*`,`*hf*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets and *hf* as hash function
Unord `c(`*bnum*`,`*hf*`,`*cmp*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets, *hf* as hash function, and *cmp* as predicate to identify equal values
Unord `c(`*c2*`)`	Copy constructor; creates a copy of another unordered container of the same type (all elements are copied)
Unord `c =` *c2*	Copy constructor; creates a copy of another unordered container of the same type (all elements are copied)
Unord `c(`*rv*`)`	Move constructor; creates an unordered container, taking the contents of the rvalue *rv* (since C++11)
Unord `c =` *rv*	Move constructor; creates an unordered container, taking the contents of the rvalue *rv* (since C++11)

Table 7.48. Possible Types *Unord* of Unordered Containers

Unord	Effect
unordered_set<*Elem*>	An unordered set that by default hashes with hash<> and compares equal_to<> (operator ==)
unordered_set<*Elem*,*Hash*>	An unordered set that by default hashes with *Hash* and compares equal_to<> (operator ==)
unordered_set<*Elem*,*Hash*,*Cmp*>	An unordered set that by default hashes with *Hash* and compares with *Cmp*
unordered_multiset<*Elem*>	An unordered multiset that by default hashes with hash<> and compares equal_to<> (operator ==)
unordered_multiset<*Elem*,*Hash*>	An unordered multiset that by default hashes with *Hash* and compares equal_to<> (operator ==)
unordered_multiset<*Elem*,*Hash*,*Cmp*>	An unordered multiset that by default hashes with *Hash* and compares with *Cmp*
unordered_map<*Key*,*T*>	An unordered map that by default hashes with hash<> and compares equal_to<> (operator ==)
unordered_map<*Key*,*T*,*Hash*>	An unordered map that by default hashes with *Hash* and compares equal_to<> (operator ==)
unordered_map<*Key*,*T*,*Hash*,*Cmp*>	An unordered map that by default hashes with *Hash* and compares with *Cmp*
unordered_multimap<*Key*,*T*>	An unordered multimap that by default hashes with hash<> and compares equal_to<> (operator ==)
unordered_multimap<*Key*,*T*,*Hash*>	An unordered multimap that by default hashes with *Hash* and compares equal_to<> (operator ==)
unordered_multimap<*Key*,*T*,*Hash*,*Cmp*>	An unordered multimap that by default hashes with *Hash* and compares with *Cmp*

For the construction, you have multiple abilities to pass arguments. On one hand, you can pass values as initial elements:

- An existing container of the same type (copy constructor)
- All elements of a range [begin,end)
- All elements of an initializer list

On the other hand, you can pass arguments that influence the behavior of the unordered container:

- The hash function (either as template or as constructor argument)
- The equivalence criterion (either as template or as constructor argument)
- The initial number of buckets (as constructor argument)

Note that you can't specify the maximum load factor as part of the type or via a constructor argument, although this is something you might often want to set initially. To specify the maximum load factor, you have to call a member function right after construction (see Table 7.49):

```
std::unordered_set<std::string> coll;
coll.max_load_factor(0.7);
```

Table 7.49. Layout Operations of Unordered Containers

Operation	Effect
`c.hash_function()`	Returns the hash function
`c.key_eq()`	Returns the equivalence predicate
`c.bucket_count()`	Returns the current number of buckets
`c.max_bucket_count()`	Returns the maximum number of buckets possible
`c.load_factor()`	Returns the current load factor
`c.max_load_factor()`	Returns the current maximum load factor
`c.max_load_factor(`*val*`)`	Sets the maximum load factor to *val*
`c.rehash(`*bnum*`)`	Rehashes the container so that it has a bucket size of at least *bnum*
`c.reserve(`*num*`)`	Rehashes the container so that is has space for at least *num* elements (since C++11)

The argument for `max_load_factor()` has to be a `float`. In general, a value between `0.7` and `0.8` provides a good compromise between speed and memory consumption. Note that the default maximum load factor is `1.0`, which means that, usually, collisions apply before rehash happens. For this reason, if speed is an issue, you should always explicitly set the maximum load factor.

Layout Operations

Unordered containers also provide operations to query and influence the internal layout. Table 7.49 lists these operations.

Besides `max_load_factor()`, the member functions `rehash()` and `reserve()` are important. They provide the functionality to rehash an unordered container (i.e., change the number of buckets) with a slightly different interface: Originally, with TR1, only `rehash()` was provided, which is a request to provide a hash table with a bucket size of at least the size passed. The problem was that with this interface, you still had to take the maximum load factor into account. If the maximum load factor is `0.7` and you want to be prepared for `100` elements, you have to divide `100` by `0.7` to compute the size that does not cause rehashing as long as no more than `100` elements have been inserted. That is, you have to pass `143` to `rehash()` to avoid further rehashing for up to `100` elements. With `reserve()`, this computing is done internally, so you can simply pass the number of elements the hash table should be prepared for:

Click here to view code image

```
coll.rehash(100);      // prepare for 100/max_load_factor() elements
coll.reserve(100);     // prepare for 100 elements (since C++11)
```

With `bucket_count()`, you can query the number of buckets an unordered container currently has. This value can be used for a number of member functions that provide a "bucket interface" you can use to inspect the exact internal state of an unordered container. See Section 7.9.4, page 374, for more details about and Section 7.9.7, page 380, for an example of inspecting the internal layout of an unordered container with the bucket interface.

Providing Your Own Hash Function

All hash tables use a hash function, which maps the values of the elements that you put into the container to a specific bucket. The goal is that two equal values always yield the same bucket index, whereas for different values, different bucket entries ideally should be processed. For any range of passed values, the hash function should provide a good distribution of hash values.

The hash function has to be a function or a function object that takes a value of the element type as parameter and returns a value of type `std::size_t`. Thus, the current number of buckets is not taken into account. Mapping the return value to the range of valid bucket indexes is done internally in the container. Thus, your goal is to provide a function that maps the different element values equally distributed in the range `[0, size_t)`.

Here is an example of how to provide your own hash function:

Click here to view code image

```
#include <functional>

class Customer {
  ...
};

class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return ...
    }
};

std::unordered_set<Customer,CustomerHash> custset;
```

Here, `CustomerHash` is a function object that defines the hash function for class `Customer`.

Instead of passing a function object to the type of the container, you can also pass a hash function as construction argument. Note, however, that the template type for the hash function must be set accordingly:

Click here to view code image

```
std::size_t customer_hash_func (const Customer& c)
{
    return ...
};

std::unordered_set<Customer,std::size_t(*)(const Customer&)>
      custset(20,customer_hash_func);
```

Here, `customer_hash_func()` is passed as second constructor argument with its type "pointer to a function taking a `Customer` and returning a `std::size_t`" passed as second template argument.

If a special hash function is not passed, the default hash function `hash<>` is used, which is provided as a function object in `<functional>` for "common" types: all integral types, all floating-point types, pointers, strings, and some special types.[15] For all other types, you have to provide your own hash function.

[15] `error_code`, `thread::id`, `bitset<>`, and `vector<bool>`

Providing a good hash function is trickier than it sounds. As a rule of thumb, you might use the default hash functions to specify your own hash functions. A naive approach would be to simply add all hash values for those attributes that are relevant for the hash function. For example:

```
class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return std::hash<std::string>()(c.fname) +
               std::hash<std::string>()(c.lname) +
               std::hash<long>()(c.no);
    }
};
```

Here, the hash value returned is just the sum of the hash value for the `Customer`'s attributes `fname`, `lname`, and `no`. If the predefined hash functions for the types of these attributes work fine for the given values, the result is the sum of three values in the range `[0, std::size_t)`. According to the common overflow rules, the resulting value should then also be reasonably well distributed.

Note, however, that experts will claim that this is still a poor hash function and that providing a good hash function can be very tricky. In addition, providing such a hash function doesn't seem as easy as it should be.

A better approach is the following, which uses a hash function provided by Boost (see [*Boost*]) and a more convenient interface:

Click here to view code image

```
// cont/hashval.hpp

#include <functional>

// from boost (functional/hash):
// see http://www.boost.org/doc/libs/1_35_0/doc/html/hash/combine.html
template <typename T>
inline void hash_combine (std::size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

// auxiliary generic functions to create a hash value using a seed
template <typename T>
inline void hash_val (std::size_t& seed, const T& val)
{
    hash_combine(seed,val);
}
template <typename T, typename... Types>
inline void hash_val (std::size_t& seed,
                      const T& val, const Types&... args)
{
    hash_combine(seed,val);
    hash_val(seed,args...);
}

// auxiliary generic function to create a hash value out of a heterogeneous list of arguments
template <typename... Types>
```

```
inline std::size_t hash_val (const Types&... args)
{
    std::size_t seed = 0;
    hash_val (seed, args...);
    return seed;
}
```

A convenience function implemented using variadic templates (see Section 3.1.9, page 26) allows calling `hash_val()` with an arbitrary number of elements of any type to process a hash value out of all these values. For example:

Click here to view code image

```
class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return hash_val(c.fname,c.lname,c.no);
    }
};
```

Internally, `hash_combine()` is called, which some experience has shown to be a good candidate for a generic hash function (see [*HoadZobel:HashCombine*]).

Especially when the input values to the hash function have specific constraints, you might want to provide your own specific hash function. In any case, to verify the effect of your own hash function, you may use the bucket interface (see Section 7.9.7, page 380).

See Section 7.9.7, page 377, for some complete examples about how to specify your own hash function. You can also use lambdas to specify the hash function (see Section 7.9.7, page 379, for details and an example).

Providing Your Own Equivalence Criterion

As the third/fourth template parameter of the unordered container types, you can pass the equivalence criterion, a predicate that is used to find equal values in the same bucket. The default predicate is `equal_to<>`, which compares the elements with operator `==`. For this reason, the most convenient approach to providing a valid equivalence criterion is to provide operator `==` for your own type if it is not predefined either as member or as global function. For example:

Click here to view code image

```
class Customer {
  ...
};
bool operator == (const Customer& c1, const Customer& c2) {
    ...
}

std::unordered_multiset<Customer,CustomerHash> custmset;
std::unordered_map<Customer,String,CustomerHash> custmap;
```

Instead, however, you can also provide your own equivalence criterion as the following example demonstrates:

Click here to view code image

```
#include <functional>

class Customer {
      ...
};

class CustomerEqual
{
  public:
    bool operator() (const Customer& c1, const Customer& c2) const {
        return  ...
    }
};

std::unordered_set<Customer,CustomerHash,CustomerEqual> custset;
std::unordered_multimap<Customer,String,
                        CustomerHash,CustomerEqual> custmmap;
```

Here, for type `Customer`, a function object is defined in which you have to implement `operator()` so that it compares two elements (or two keys for maps) and returns a Boolean value indicating whether they are equal.

See Section 7.9.7, page 377, for a complete example about how to provide your own equivalence criterion (and hash function). Again, you can also use lambdas to specify the equivalence criterion (see Section 7.9.7, page 379, for details and an example).

Whenever values are considered to be equal according to the current equivalence criterion, they should also yield the same hash values according to the current hash function. For this reason, an unordered container that is instantiated with a nondefault equivalence predicate usually needs a nondefault hash function as well.

7.9.3. Other Operations for Unordered Containers

The remaining operations for unordered containers are more or less the same as for associative containers.

Nonmodifying Operations

Unordered containers provide the nonmodifying operations listed in Table 7.50.

Table 7.50. Nonmodifying Operations of Unordered Containers

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
c1 `==` *c2*	Returns whether *c1* is equal to *c2*
c1 `!=` *c2*	Returns whether *c1* is not equal to *c2* (equivalent to `!(`*c1*`==`*c2*`)`)

Note again that for comparisons, only operators `==` and `!=` are provided for unordered containers.[16]

[16] Operators `==` and `!=` are not provided for unordered containers with TR1.

In worst-case scenarios, they might, however, provide quadratic complexity.

Special Search Operations

Unordered containers are optimized for fast searching of elements with a specific value (unordered sets and multisets) or key (unordered maps and multimaps). To benefit from this behavior, the containers provide special search functions (see Table 7.51). These functions are special versions of general algorithms that have the same name. You should always prefer the optimized versions for unordered containers to achieve constant complexity instead of the linear complexity of the general algorithms, provided that the hash values are evenly distributed. For example, a search in a collection of 1,000 elements requires on average only 1 comparison instead of 10 for associative containers and 500 for sequence containers (see Section 2.2, page 10).

Table 7.51. Special Search Operations of Unordered Containers

Operation	Effect
`c.count(`*val*`)`	Returns the number of elements with key/value *val*
`c.find(`*val*`)`	Returns the position of the first element with key/value *val* (or `end()` if none found)
`c.equal_range(`*val*`)`	Returns a range with all elements with a key/value equal to *val* (i.e., the first and last positions, where `val` would get inserted)

See Section 7.7.2, page 319, for a detailed description of these member functions, including an example program that demonstrates their application.

Assignments

As listed in Table 7.52, unordered containers provide only the fundamental assignment operations that all containers provide (see Section 7.1.2, page 258).

Table 7.52. Assignment Operations of Unordered Containers

Operation	Effect
`c =` *c2*	Assigns all elements of *c2* to `c`
`c =` *rv*	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c =` *initlist*	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
c1`.swap(`*c2*`)`	Swaps the data of *c1* and *c2*
`swap(`*c1*`,`*c2*`)`	Swaps the data of *c1* and *c2*

For these operations, both containers must have the same type. In particular, the type of the hash functions and the equivalence criteria must be the same, although the functions themselves may be different. If the functions are different, they will also get assigned or swapped.

Iterator Functions and Element Access

Unordered containers do not provide direct element access, so you have to use range-based `for` loops (see Section 3.1.4, page 17) or iterators. Because the iterators are guaranteed to be only forward iterators (see Section 9.2.3, page 436), no support for bidirectional iterators

or random-access iterators is provided (Table 7.53). Thus, you can't use them in algorithms that are provided for bidirectional iterators or random-access iterators only, such as algorithms for sorting or random shuffling.

Table 7.53. Iterator Operations of Unordered Containers

Operation	Effect
`c.begin()`	Returns a forward iterator for the first element
`c.end()`	Returns a forward iterator for the position after the last element
`c.cbegin()`	Returns a constant forward iterator for the first element (since C++11)
`c.cend()`	Returns a constant forward iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

For unordered (multi)sets, all elements are considered constant from an iterator's point of view. For unordered (multi)maps, the key of all elements is considered to be constant. This is necessary to ensure that you can't compromise the position of the elements by changing their values. Although there is no specific order, the value defines the bucket position according to the current hash function. For this reason, you can't call any modifying algorithm on the elements. For example, you can't call the `remove()` algorithm, because it "removes" by overwriting "removed" elements with the following elements (see Section 6.7.2, page 221, for a detailed discussion of this problem). To remove elements in unordered sets and multisets, you can use only member functions provided by the container.

In correspondence to maps and multimaps, the type of the elements of unordered maps and multimap is `pair<const Key, T>`, which means that you need `first` and `second` to access the key and the value of an element (see Section 7.8.2, page 337 for details):

Click here to view code image

```
std::unordered_map<std::string,float> coll;
...
for (auto& elem : coll) {
    std::cout << "key: "   << elem.first  << "\t"
              << "value: " << elem.second << std::endl;
}
...
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << "key: "   << pos->first  << "\t"
              << "value: " << pos->second << std::endl;
}
```

Trying to change the value of the key results in an error:

```
elem.first = "hello";    // ERROR at compile time
pos->first = "hello";    // ERROR at compile time
```

However, changing the value of the element is no problem, as long as `elem` is a nonconstant reference and the type of the value is not constant:

```
elem.second = 13.5;      // OK
pos->second = 13.5;      // OK
```

If you use algorithms and lambdas to operate on the elements of a map, you explicitly have to declare the element type:

Click here to view code image

```
std::unordered_map<std::string,int> coll;
...
// add 10 to the value of each element:
std::for_each (coll.begin(), coll.end(),
               [] (std::pair<const std::string,int>& elem) {
                   elem.second += 10;
               });
```

Instead of using the following:

```
std::pair<const std::string,int>
```

you could also use

```
std::unordered_map<std::string,int>::value_type
```

or

```
decltype(coll)::value_type
```

to declare the type of an element. See Section 7.8.5, page 345, for a complete example with maps that also works with unordered maps.

To change the key of an element, you have only one choice: You must replace the old element with a new element that has the same value. This is described for maps in Section 7.8.2, page 339.

Note that unordered maps can also be used as associative arrays, so you can use the subscript operator to access elements. See Section 7.9.5, page 374, for details.

Note also that there is an additional iterator interface to iterate over the buckets of an unordered container. See Section 7.9.7, page 380, for details.

Inserting and Removing Elements

Table 7.54 shows the operations provided for unordered containers to insert and remove elements.

Table 7.54. Insert and Remove Operations of Unordered Containers

Operation	Effect
`c.insert(`*val*`)`	Inserts a copy of *val* and returns the position of the new element and, for `unordered_sets` and `unordered_maps`, whether it succeeded
`c.insert(`*pos*`,`*val*`)`	Inserts a copy of *val* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.insert(`*beg*`,`*end*`)`	Inserts a copy of all elements of the range [*beg*,*end*) (returns nothing)
`c.insert(`*initlist*`)`	Inserts a copy of all elements in the initializer list *initlist* (returns nothing; since C++11)
`c.emplace(`*args*`...)`	Inserts a new element initialized with *args* and returns the position of the new element and, for `unordered_sets` and `unordered_maps`, whether it succeeded (since C++11)
`c.emplace_hint(`*pos*`,`*args*`...)`	Inserts a new element initialized with *args* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.erase(`*val*`)`	Removes all elements equal to *val* and returns the number of removed elements
`c.erase(`*pos*`)`	Removes the element at iterator position *pos* and returns

As usual when using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, and the beginning of a range must have a position that is not behind the end.

In general, erasing functions do not invalidate iterators and references to other elements. However, the `insert()` and `emplace()` members may invalidate all iterators when rehashing happens, whereas references to elements always remain valid. Rehashing happens when, because of an insertion, the number of resulting elements is equal to or exceeds the bucket count times the maximum load factor (i.e., when the guarantee given by the maximum load factor would be broken). Note again that all `erase()` operations never cause rehashing so that it is guaranteed that iterators, references, and pointers to other elements remain valid. The `insert()` and `emplace()` members do not affect the validity of references to container elements.

Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls. Note, however, that the exception guarantees are reduced for multi-element operations (see Section 7.9.6, page 375).

Note that the return types of the inserting functions `insert()` and `emplace()` differ as follows:

- **Unordered sets** provide the following interface:

Click here to view code image

```
pair<iterator,bool> insert(const value_type&  val);
iterator            insert(iterator  posHint,
                           const value_type&  val);
template <typename... Args>
pair<iterator,bool> emplace(Args&&... args);
template <typename... Args>
iterator            emplace_hint(const_iterator  posHint,
                                 Args&&...  args);
```

- **Unordered multisets** provide the following interface:

Click here to view code image

```
iterator            insert(const value_type&  val);
iterator            insert(iterator  posHint,
                           const value_type&  val);
template <typename... Args>
iterator            emplace(Args&&...  args);
template <typename... Args>
iterator            emplace_hint(const_iterator  posHint,
                                 Args&&...  args);
```

The difference in return types results because unordered multisets and multimaps allow duplicates, whereas unordered sets and maps do not. Thus, the insertion of an element might fail for an unordered set if it already contains an element with the same value. Therefore, the return type for a set returns two values by using a `pair` structure (`pair` is discussed in Section 5.1.1, page 60):

1. The member `second` of the `pair` structure returns whether the insertion was successful.
2. The member `first` of the `pair` structure returns the position of the newly inserted element or the position of the still existing element.

In all other cases, the functions return the position of the new element or of the existing element if the unordered set already contains an element with the same value. See Section 7.7.2, page 323, for some examples about these interfaces.

For unordered maps and multimaps, to insert a key/value pair, you must keep in mind that inside, the key is considered to be constant. You must provide either the correct type or implicit or explicit type conversions.

Since C++11, the most convenient way to insert elements is to use `emplace()` or to pass them to `insert()` as initializer list, where the first entry is the key and the second entry is the value:

```
std::unordered_map<std::string,float> coll;
...
coll.emplace("jim",17.7);   // see below, if key/value need more args for initialization
coll.insert({"otto",22.3});
```

Alternatively, there are also the three other ways to pass a value into an unordered map or multimap that were introduced for maps and multimaps already (see Section 7.8.2, page 341, for details):

1. Use `value_type` :

Click here to view code image

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::unordered_map<std::string,float>::value_type
                ("otto",22.3));
coll.insert(decltype(coll)::value_type("otto",22.3));
```

2. Use `pair<>` :

Click here to view code image

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::pair<std::string,float>("otto",22.3));
coll.insert(std::pair<const std::string,float>("otto",22.3));
```

3. Use `make_pair()` :

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

Here is a simple example of the insertion of an element into an unordered map that also checks whether the insertion was successful:

Click here to view code image

```
std::unordered_map<std::string,float> coll;
```

```
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

Note again that when using `emplace()` and the key and/or the element value require more than one value for initialization, you have to pass two tuples of arguments: one for the key and one for the value. The most convenient way to do this is as follows:

Click here to view code image

```
std::unordered_map<std::string,std::complex<float>> m;

m.emplace(std::piecewise_construct,      // pass tuple elements as arguments
          std::make_tuple("hello"),      // elements for the key
          std::make_tuple(3.4,7.8));     // elements for the value
```

See Section 5.1.1, page 63, for details of piecewise construction of pairs.

Note again that unordered maps provide another convenient way to insert and set elements with the subscript operator (see Section 7.9.5, page 374).

To remove an element that has a certain value, you simply call `erase()` :

```
std::unordered_set<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

Note that this member function has a different name than `remove()` provided for lists (see Section 7.5.2, page 294, for a discussion of `remove()`). It behaves differently because it returns the number of removed elements. When called for unordered maps, it returns only `0` or `1` .

If an unordered multiset or multimap contains duplicates and you want to remove only the first element of these duplicates, you can't use `erase()` . Instead, you can code as follows:

```
std::unordered_multimap<Key,T> coll;
...
// remove first element with passed value
auto pos = coll.find(value);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

You should use the member function `find()` here because it is faster than the `find()` algorithm (see the example in Section 7.3.2, page 277).

When removing elements, be careful not to saw off the branch on which you are sitting. See Section 7.8.2, page 342, for a detailed description of this problem.

7.9.4. The Bucket Interface

It is possible to access the individual buckets with a specific bucket interface to expose the internal state of the whole hash table. Table 7.55 shows the operations provided to directly access buckets.

Table 7.55. Bucket Interface Operations of Unordered Sets and Multisets

Operation	Effect
`c.bucket_count()`	Returns the current number of buckets
`c.bucket(`*val*`)`	Returns the index of the bucket in which *val* would/could be found
`c.bucket_size(`*buckidx*`)`	Returns the number of elements in the bucket with index *buckidx*
`c.begin(`*buckidx*`)`	Returns a forward iterator for the first element of the bucket with index *buckidx*
`c.end(`*buckidx*`)`	Returns a forward iterator for the position after the last element of the bucket with index *buckidx*

See Section 7.9.7, page 380, for an example of how to use the bucket interface to inspect the internal layout of an unordered container.

7.9.5. Using Unordered Maps as Associative Arrays

As with maps, unordered maps provide a subscript operator for direct element access and a corresponding member function `at()` (see Table 7.56).

Table 7.56. Direct Element Access of Unordered Maps

Operation	Effect
`c[`*key*`]`	Inserts an element with *key*, if it does not yet exist, and returns a reference to the value of the element with *key* (only for nonconstant unordered maps)
`c.at(`*key*`)`	Returns a reference to the value of the element with *key* (since C++11)

`at()` yields the value of the element with the passed key and throws an exception object of type `out_of_range` if no such element is present.

For operator `[]`, the index also is the key used to identify the element. This means that for operator `[]`, the index may have any type rather than only an integral type. Such an interface is the interface of a so-called *associative array*.

For operator `[]`, the type of the index is not the only difference from ordinary C-style arrays. In addition, you can't have a wrong index. If you use a key as the index for which no element yet exists, a new element gets inserted into the map automatically. The value of the new element is initialized by the default constructor of its type. Thus, to use this feature, you can't use a value type that has no default constructor. Note that the fundamental data types provide a default constructor that initializes their values to zero (see Section 3.2.1, page 37).

See Section 7.8.3, page 344, for a detailed discussion of the advantages and disadvantages this container interface provides. See Section 6.2.4, page 185, and Section 7.8.5, page 346, for some example code (partially using maps, which provide the same interface).

7.9.6. Exception Handling

Unordered containers are node-based containers, so any failure to construct a node simply leaves the container as it was. However, the fact that a rehashing may occur comes into account. For this reason, the following guarantees apply to all unordered containers:

- Single-element insertions have the commit-or-rollback behavior, provided that the hash and equivalence functions don't throw. Thus, if they don't throw, the operations either succeed or have no effect.
- `erase()` does not throw an exception, provided that the hash function and the equivalence criterion don't throw, which is the case for the default functions.
- No `clear()` function throws an exception.
- No `swap()` function throws an exception, provided that the copy constructor or the copy assignment operator of the hash or equivalence functions don't throw.
- `rehash()` has the commit-or-rollback behavior, provided that the hash and equivalence functions don't throw. Thus, if they don't throw, the operations either succeed or have no effect.

See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

7.9.7. Examples of Using Unordered Containers

The following program demonstrates the fundamental abilities of unordered containers, using an unordered set:

Click here to view code image

```
// cont/unordset1.cpp

#include <unordered_set>
#include <numeric>
#include "print.hpp"
using namespace std;

int main()
{
    // create and initialize unordered set
    unordered_set<int> coll = { 1,2,3,5,7,11,13,17,19,77 };

    // print elements
    // - elements are in arbitrary order
    PRINT_ELEMENTS(coll);

    // insert some additional elements
    // - might cause rehashing and create different order
    coll.insert({-7,17,33,-11,17,19,1,13});
    PRINT_ELEMENTS(coll);

    // remove element with specific value
    coll.erase(33);

    // insert sum of all existing values
    coll.insert(accumulate(coll.begin(),coll.end(),0));
    PRINT_ELEMENTS(coll);

    // check if value 19 is in the set
    if (coll.find(19) != coll.end()) {
        cout << "19 is available" << endl;
    }

    // remove all negative values
    unordered_set<int>::iterator pos;
    for (pos=coll.begin(); pos!= coll.end(); ) {
        if (*pos < 0) {
            pos = coll.erase(pos);
        }
        else {
            ++pos;
        }
    }
    PRINT_ELEMENTS(coll);
}
```

As long as you only insert, erase, and find elements with a specific value, unordered containers provide the best running-time behavior because all these operations have amortized constant complexity. However, you can't make any assumption about the order of the elements. For example, the program might have the following output:

```
77 11 1 13 2 3 5 17 7 19
-11 1 2 3 -7 5 7 77 33 11 13 17 19
-11 1 2 3 -7 5 7 77 11 13 17 19 137
19 is available
1 2 3 5 7 77 11 13 17 19 137
```

For anything else – for example, to accumulate the values in the container or find and remove all negative values — you have to iterate over all elements (either directly with iterators or indirectly using a range-based `for` loop).

When using an unordered multiset rather than an unordered set, duplicates are allowed. For example, the following program:

Click here to view code image

```
// cont/unordmultiset1.cpp

#include <unordered_set>
#include "print.hpp"
using namespace std;

int main()
{
    // create and initialize, expand, and print unordered multiset
    unordered_multiset<int> coll = { 1,2,3,5,7,11,13,17,19,77 };
    coll.insert({-7,17,33,-11,17,19,1,13});
    PRINT_ELEMENTS(coll);

    // remove all elements with specific value
    coll.erase(17);

    // remove one of the elements with specific value
```

```
    auto pos = coll.find(13);
    if (pos != coll.end()) {
        coll.erase(pos);
    }
    PRINT_ELEMENTS(coll);
}
```

might have the following output:

```
33 19 19 17 17 17 77 11 7 -7 5 3 13 13 2 -11 1 1
33 19 19 77 11 7 -7 5 3 13 2 -11 1 1
```

Example of Providing Your Own Hash Function and Equivalence Criterion

The following example shows how to define and specify a hash function and an equivalence criterion for a type `Customer`, which is used as element type of an unordered set:

Click here to view code image

```
// cont/hashfunc1.cpp

#include <unordered_set>
#include <string>
#include <iostream>
#include "hashval.hpp"
#include "print.hpp"

using namespace std;

class Customer {
  private:
    string fname;
    string lname;
    long   no;
  public:
    Customer (const string& fn, const string& ln, long n)
      : fname(fn), lname(ln), no(n) {}
    friend ostream& operator << (ostream& strm, const Customer& c) {
        return strm << "[" << c.fname << "," << c.lname << ","
                           << c.no << "]";
    }
    friend class CustomerHash;
    friend class CustomerEqual;
};

class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return hash_val(c.fname,c.lname,c.no);
    }
};

class CustomerEqual
{
  public:
    bool operator() (const Customer& c1, const Customer& c2) const {
        return c1.no == c2.no;
    }
};

int main()
{
    // unordered set with own hash function and equivalence criterion
    unordered_set<Customer,CustomerHash,CustomerEqual> custset;

    custset.insert(Customer("nico","josuttis",42));
    PRINT_ELEMENTS(custset);

}
```

The program has the following output:

```
[nico,josuttis,42]
```

Here, the `hash_val()` convenience function for an arbitrary number of elements of different types introduced in Section 7.9.2, page 364, is used.

As you can see, the equivalence function does not necessarily have to evaluate the same values as the hash function. However, as written, it should be guaranteed that values that are equal according the equivalence criterion yield the same hash value (which indirectly is the case

here assuming that customer numbers are unique).

Without specifying an equivalence function, the declaration of `custset` would be:

```
std::unordered_set<Customer,CustomerHash> custset;
```

and operator `==` would be used as equivalence criterion, which you had to define for `Customer` s instead.

You could also use an ordinary function as hash function. But in that case you have to pass the function as constructor argument, which means that you also have to pass the initial bucket count and specify the corresponding function pointer as second template parameter (see Section 7.9.2, page 363, for details).

Using Lambdas as Hash Function and Equivalence Criterion

You can even use lambdas to specify the hash function and/or the equivalence criterion. For example:

Click here to view code image

```
// cont/hashfunc2.cpp

#include <string>
#include <iostream>
#include <unordered_set>
#include "hashval.hpp"
#include "print.hpp"
using namespace std;

class Customer {
  private:
    string fname;
    string lname;
    long   no;
  public:
    Customer (const string& fn, const string& ln, long n)
      : fname(fn), lname(ln), no(n) {
    }
    string firstname() const {
        return fname;
    };

    string lastname() const {
        return lname;
    };
    long number() const {
        return no;
    };
    friend ostream& operator << (ostream& strm, const Customer& c) {
        return strm << "[" << c.fname << "," << c.lname << ","
                           << c.no << "]";
    }
};

int main()
{
    // lambda for user-defined hash function
    auto hash = [] (const Customer& c) {
        return hash_val(c.firstname(),c.lastname(),c.number());
    };

    // lambda for user-defined equality criterion
    auto eq = [] (const Customer& c1, const Customer& c2) {
        return c1.number() == c2.number();
    };

    // create unordered set with user-defined behavior
    unordered_set<Customer,
                  decltype(hash),decltype(eq)> custset(10,hash,eq);

    custset.insert(Customer("nico","josuttis",42));
    PRINT_ELEMENTS(custset);

}
```

Note that you have to use `decltype` to yield the type of the lambda to be able to pass it as template argument to the declaration of the unordered container. The reason is that for lambdas, no default constructor and assignment operator are defined. Therefore, you also have to pass the lambdas to the constructor. This is possible only as second and third arguments. Thus, you have to specify the initial bucket size `10` in this case.

Example of Using the Bucket Interface

The following example demonstrates an application of the bucket interface provided to inspect the internal state of an unordered container (see Section 7.9.4, page 374). In `printHashTableState()`, the whole state, including the detailed layout of an unordered container, is printed:

Click here to view code image

```
// cont/buckets.hpp

#include <iostream>
#include <iomanip>
#include <utility>
#include <iterator>
#include <typeinfo>

// generic output for pairs (map elements)
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm, const std::pair<T1,T2>&
p)
{
    return strm << "[" << p.first << "," << p.second << "]";
}

template <typename T>
void printHashTableState (const T& cont)
{
    // basic layout data:
    std::cout << "size:            " << cont.size() << "\n";
    std::cout << "buckets:         " << cont.bucket_count() << "\n";
    std::cout << "load factor:     " << cont.load_factor() << "\n";
    std::cout << "max load factor: " << cont.max_load_factor() << "\n";

    // iterator category:
    if (typeid(typename std::iterator_traits
                          <typename T::iterator>::iterator_category)
        == typeid(std::bidirectional_iterator_tag)) {
          std::cout << "chaining style:   doubly-linked" << "\n";
    }
    else {
          std::cout << "chaining style:   singly-linked" << "\n";
    }

    // elements per bucket:
    std::cout << "data: " << "\n";
    for (auto idx=0; idx != cont.bucket_count(); ++idx) {
            std::cout << " b[" << std::setw(2) << idx << "]: ";
            for (auto pos=cont.begin(idx); pos != cont.end(idx); ++pos)
{
                std::cout << *pos << " ";
            }
            std::cout << "\n";
    }
    std::cout << std::endl;
}
```

For example, you can use this header file to print the internal layout of an unordered set:

Click here to view code image

```
// cont/unordinspect1.cpp

#include <unordered_set>
#include <iostream>
#include "buckets.hpp"

int main()
{
    // create and initialize an unordered set
    std::unordered_set<int> intset = { 1,2,3,5,7,11,13,17,19 };
    printHashTableState(intset);

    // insert some additional values (might cause rehashing)
    intset.insert({-7,17,33,4});
    printHashTableState(intset);
}
```

Comparing the first and second call of `printHashTableState()`, the program might have the following output (details depend on the concrete layout and rehashing strategy of the standard library used):

```
size:       9          |  size:       12
buckets:    11         |  buckets:    23
```

As another example for the application of the bucket interface, the following program creates a dictionary of string values mapped to other string values (compare this example with a corresponding version for maps in Section 7.8.5, page 348):

```
// cont/unordmultimap1.cpp

#include <unordered_map>
#include <string>
#include <iostream>
#include <utility>
#include "buckets.hpp"
using namespace std;

int main()
{
    // create and initialize an unordered multimap as dictionary
    std::unordered_multimap<string,string> dict = {
                  {"day","Tag"},
                  {"strange","fremd"},
                  {"car","Auto"},
                  {"smart","elegant"},
                  {"trait","Merkmal"},
                  {"strange","seltsam"}
    };
    printHashTableState(dict);

    // insert some additional values (might cause rehashing)
    dict.insert({{"smart","raffiniert"},
                 {"smart","klug"},
                 {"clever","raffiniert"}
                });
    printHashTableState(dict);

    // modify maximum load factor (might cause rehashing)
    dict.max_load_factor(0.7);
    printHashTableState(dict);
}
```

Again, the output of this program is implementation specific. For example, the output of this program might be as follows (slightly modified to fit the page width):

Whereas on another platform the output of this program might be as follows (again, slightly modified to fit the page width):

Note that in any case rehashing preserves the relative ordering of equivalent elements. However, the order of equivalent elements might not match the order of their insertion.