**Unordered Associative Containers**

With the hash table-based implementations, the elements have no defined order.

Unordered associative containers can be accessed 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.

**Differences of individual classes:**

1. 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.
2. Unordered sets and maps allow no duplicates, whereas unordered multisets and multimaps do.

header file

for unordered set or multiset #include <unordered\_set>

for unordered map or multimap #include <unordered\_map>

The types are defined as class templates inside namespace std:

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;

}

namespace std {

template < typename Key, typename T,

typename Hash = hash<T>,

typename EqPred = equal\_to<T>,

typename Allocator = allocator<pair<const Key, T> > >

class unordered\_map;

template < typename Key, typename T,

typename Hash = hash<T>,

typename EqPred = equal\_to<T>,

typename Allocator = allocator<pair<const Key, T> > >

class unordered\_multimap;

}

**T: element type**

* The elements of an unordered set or multiset may have any type T that is comparable.
* The elements of an unordered map or an unordered multimap may have any types Key and T that meet the following two requirements:
  + Both key and value must be copyable or movable.
  + The key must be comparable with the equivalence criterion.

The element type (value\_type) is a pair<const Key,T>.

**Hash: Hash function**

* optional parameter that defines the hash function
* 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 like error\_code, thread::id, bitset<>, and vector<bool>

* For all other value types, you must pass your own hash function.

**EqPred: equivalence criterion**

* Defines an equivalence criterion: a predicate that is used to find elements.
* Default criterion equal\_to<> is used

**Allocator: memory model allocator**

* defines the memory model
* Default memory model is the model allocator

# Abilities of Unordered Containers

All standardized unordered container classes are implemented as hash tables, which nonetheless still have a variety of implementation options.

The specified abilities of unordered containers are based on the following assumptions:

* 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".)
* 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

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

The major advantage of using a hash table internally is its incredible running-time behavior. With well-chosen and well implemented hashing strategy, you can guarantee amortized constant time for insertions, deletions, and element search.

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, 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.

**Disadvantages over ordinary associative containers:**

1. Unordered containers don’t provide operators <, >, <=, and >= to order multiple instances of these containers.

However, == and != are provided (since C++11).

1. lower\_bound() and upper\_bound() are not provided.
2. Because the iterators are guaranteed only to be forward iterators, reverse iterators are not supported, and you can’t use algorithms that require bidirectional iterators, or at least this is not portable.

You are not allowed to modify the (key) value of an element directly because the (key) value of an element specifies its position — in this case, its bucket entry.

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:

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

**Parameters that influence the behavior of the hash table:**

1. You can specify the minimum number of buckets.
2. You can (and sometimes have to) provide your own hash function.
3. 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.
4. You can specify a maximum load factor, which leads to automatic rehashing when it is exceeded.
5. You can force rehashing.

You **can’t influence** the following behavior:

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

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 the 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.

# Creating and Controlling Unordered Containers

## Create, Copy, and Destroy

List of the constructors and destructors of unordered associative 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) |
| Unord c(beg,end) | Creates an unordered container initialized by the elements of the range [beg,end) |
| Unord c(beg,end,bnum) | Creates an unordered container initialized by the elements of the range [beg,end) that internally uses at least bnum buckets |
| Unord c(beg,end,bnum,hf) | Creates an unordered container initialized by the elements of the range [beg,end) that internally uses at least bnum buckets and hf as hash function |
| Unord c(beg,end,bnum,hf,cmp) | Creates an unordered container initialized by the elements of the range [beg,end) that internally uses at least bnum buckets, hf as hash function, and cmp as predicate to identify equal values |
| Unord c(initlist) | Creates an unordered unordered container initialized by the elements of the initializer list initlist |
| Unord c = initlist | Creates an unordered unordered container initialized by the elements of the initializer list initlist |
| c.~Unord() | Destroys all elements and frees the memory |

List of the types Unord that can be used with these constructors and destructors.

|  |  |
| --- | --- |
| Operation | 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 can pass values as initial elements:

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

You can pass arguments that influence the behavior of the unordered container:

1. The hash function (either as template or as constructor argument)
2. The equivalence criterion (either as template or as constructor argument)
3. The initial number of buckets (as constructor argument)

You can’t specify the maximum load factor as part of the type or via a constructor argument. To specify the maximum load factor, you have to call a member function right after construction

std::unordered\_set<std::string> coll;

coll.max\_load\_factor(0.7); // float value

Default maximum load factor is 1.0, which means that, usually, collisions apply before rehash happens. A value between 0.7 and 0.8 provides a good compromise between speed and memory consumption.

## Layout Operations

|  |  |
| --- | --- |
| 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) |

**rehash():** To provide a hash table with a bucket size of at least the size passed.

Drawback - with this interface, you still had to take the maximum load factor into account.

**reserve():** You can simply pass the number of elements the hash table should be prepared for, and maximum load factor computing is done internally.

coll.rehash(100); // prepare for 100/max\_load\_factor() elements

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

## Providing Your Own Hash Function

A hash function maps the values of the elements that you put into the container to a specific bucket. 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.

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 floatingpoint types, pointers, strings, and some special types error\_code, thread::id, bitset<>, and vector<bool>.

For all other types, you have to provide your own hash function.

### Function Object

#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.

### Hash Function

You can also pass a hash function as construction argument. The template type for the hash function must be set accordingly:

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.

You can also use lambdas to specify the hash function.

## Providing Your Own Equivalence Criterion

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 ==.

Provide operator == for your own type if it is not predefined either as member or as global function.

Example:

class Customer { ... };

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

std::unordered\_multiset<Customer, CustomerHash> custmset;

std::unordered\_map<Customer, String, CustomerHash> custmap;

You can also provide your own equivalence criterion as the following example demonstrates:

#include <functional>

class Customer { ... };

class CustomerEqual {

public:

bool operator() (const Customer& c1, Customer& c2) const { return ... }

};

std::unordered\_set<Customer, CustomerHash, CustomerEqual> custset;

std::unordered\_multimap<Customer, String, CustomerHash, CustomerEqual> custmmap;

You can also use lambdas to specify the equivalence criterion.

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.

# Other Operations

## Nonmodifying Operations

|  |  |
| --- | --- |
| 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)) |

For comparisons, only operators == and != are provided for unordered containers. (Operators == and != are not provided for unordered containers with TR1.)

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

## Special Search Operations

|  |  |
| --- | --- |
| Operation | Effect |
| c.count(val) | Returns the number of elements with value val |
| c.find(val) | Returns the position of the first element with value val (or end() if none found) |
| c.equal\_range(val) | Returns a range with all elements with a value equal to |

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.

## Assignments

|  |  |
| --- | --- |
| 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

|  |  |
| --- | --- |
| 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) |

Unordered containers do not provide direct element access.

Because the iterators are guaranteed to be only forward iterators no support for bidirectional iterators or random-access iterators is provided. Thus, you can’t use algorithms for sorting or random shuffling.

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.

The type of the elements of unordered maps and multimap is pair<const Key, T>.

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:

std::unordered\_map<std::string,int> coll;

std::for\_each (coll.begin(), coll.end(),

[] (pair<const std::string,int>& elem) { elem.second += 10;});

Instead of using the following:

pair<const std::string,int>

you could also use

unordered\_map<std::string,int>::value\_type

or

decltype(coll)::value\_type

To change the key of an element, replace the old element with a new element that has the same value.

## Inserting and Removing Elements

|  |  |
| --- | --- |
| Operation | Effect |
| c.insert(val) | Inserts a copy of val and returns the position of the new element and, for unordered containers, 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 copy of an element initialized with args and returns the position of the new element and, for unordered containers, whether it succeeded (since C++11) |
| c.emplace\_hint(pos, args...) | Inserts a copy of an 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 the following position (returned nothing before C++11) |
| c.erase(beg, end) | Removes all elements of the range [beg,end) and returns the following position (returned nothing before C++11) |
| c.clear() | Removes all elements (empties the container) |

Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

In general, erasing functions do not invalidate iterators and references to other elements.

### Insertion

The insert() and emplace() members may invalidate all iterators when rehashing happens, whereas references to elements always remain valid.

Return types of the inserting functions insert() and emplace() differ as follows:

**Unordered sets:**

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:**

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);

Since C++11, the most convenient way to insert elements is to pass them as an initializer list:

std::unordered\_map<std::string,float> coll;

coll.insert({"otto", 22.3});

Three other ways to pass a value into an unordered map or multimap:

1. **Use value\_type**

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));

1. **Use pair<>**

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));

1. **Use make\_pair()**

std::unordered\_map<std::string, float> coll;

coll.insert(std::make\_pair("otto", 22.3));

Unordered maps provide another convenient way to insert and set elements with the subscript operator [].

When using emplace() to insert a new element by passing the values for its construction, you have to pass two lists of arguments.

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

### Remove

To remove an element call erase():

std::unordered\_set<Elem> coll;

…

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

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 use:

std::unordered\_multimap<Key,T> coll;

...

auto pos = coll.find(value); // find first element with passed value

if (pos != coll.end()) {

coll.erase(pos); // remove first element with passed value

}

When removing elements, be careful not to saw off the branch on which you are sitting.

# 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.

|  |  |
| --- | --- |
| 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 |
| c.cbegin(buckidx) | Returns a constant forward iterator for the first element of the bucket with index buckidx |
| c.cend(buckidx) | Returns a constant forward iterator for the position after the last element of the bucket with index buckidx |

# Using Unordered Maps as Associative Arrays

Unordered maps provide a subscript operator for direct element access and a corresponding member function at().

|  |  |
| --- | --- |
| 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) |

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.

# Exception Handling

Unordered containers are node-based containers, so any failure to construct a node simply leaves the container as it was.

1. 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.
2. **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.
3. No **clear()** function throws an exception.
4. 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.
5. **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.

# Example

#include <iostream>

#include <unordered\_set>

using namespace std;

int main () {

unordered\_set<int> coll = { 1,2,3,5,7,7,11,13,17,19,77 };

unordered\_set<int> coll1 = { 1,2,3,5,7,11,13,17,19,77 };

unordered\_multiset<int> coll2 = { 1,2,3,5,7,7,11,13,17,19,77 };

unordered\_set<int>::iterator pos;

for (pos=coll.begin(); pos!= coll.end(); ++pos) {

cout << \*pos << ",";

}

cout << endl;

coll.insert(7);

coll.erase(13);

for (pos=coll.begin(); pos!= coll.end(); ++pos) {

cout << \*pos << ",";

}

cout << endl;

return 0;

}

Output:

77,19,17,13,11,7,5,3,2,1,

77,19,17,11,7,5,3,2,1,

## Example of Providing Your Own Hash Function and Equivalence Criterion

/Header File cppheader.h/

#include <functional>

template <typename T>

inline void hash\_combine (std::size\_t& seed, const T& val) {

seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);

}

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...);

}

template <typename... Types>

inline std::size\_t hash\_val (const Types&... args) {

std::size\_t seed = 0;

hash\_val (seed, args...);

return seed;

}

/CPP file/

#include <unordered\_set>

#include <string>

#include <iostream>

#include "cppheader.h"

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, 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("ratnesh", "tiwari", 555));

unordered\_set<Customer, CustomerHash, CustomerEqual>::iterator pos;

for (pos=custset.begin(); pos!= custset.end(); ++pos) {

cout << \*pos << ",";

}

cout << endl;

return 0;

}

Output:

[ratnesh, kumar, 555],

## Using Lambdas as Hash Function and Equivalence Criterion

#include <unordered\_set>

#include <string>

#include <iostream>

#include "cppheader.h"

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, 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("ratnesh", "kumar", 555));

unordered\_set<Customer, decltype(hash), decltype(eq)>::iterator pos;

for (pos=custset.begin(); pos!= custset.end(); ++pos) {

cout << \*pos << ",";

}

cout << endl;

return 0;

}

Output:

[ratnesh, kumar, 555],

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.

# END