MODULE 29
--THE STL--
CONTAINER PART III
map, multimap, hash_map, hash_multimap, hash_set, hash_multiset

My Training Period: hours

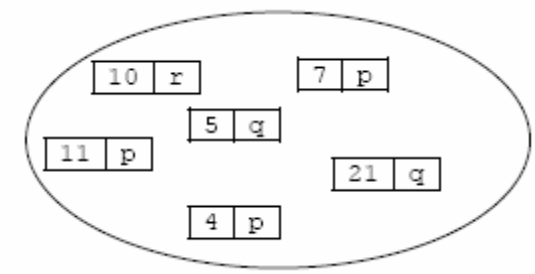
Note:
Compiled using VC++7.0 / .Net, win32 empty console mode application. Be careful with the source codes than span more than one line. **g++** compilation examples are given at the end of this Module.

Abilities

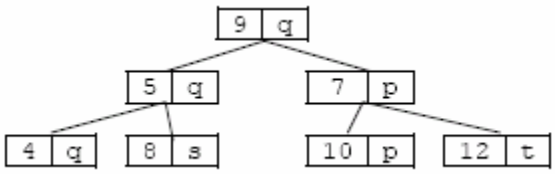
- Able to understand and use `map` associative container.
- Able to understand and use `multimap` associative container.
- Able to understand and use `hash_map` associative container.
- Able to understand and use `hash_multimap` associative container.
- Able to understand and use `hash_set` associative container.
- Able to understand and use `hash_multiset` container.
- Remember some useful summary.

29.1 map

- A map contains elements that are **key** and **value pairs**. Each element has a key that is the basis for the sorting criterion and a value.
- Each key may occur only once, thus duplicate keys are not allowed.
- A map can also be used as an **associative array**, which is an array that has an arbitrary index type. It can be depicted as follow:



- The binary tree of the map and multimap structure can be depicted as follow:



- The iterator provided by the map class is a bidirectional iterator, but the class member functions `insert()` and `map()` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators.
- The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator.
- This type of structure is an ordered list of **uniquely occurring key words** with associated **string values**. If, instead, the words had more than one correct definition, so that keys were not unique, then a multimap would be the container of choice.
- If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure.

- The map orders the sequence it controls by calling a stored function object of type `key_compare`. This stored object is a comparison function that may be accessed by calling the member function `key_comp()`.
- The general format of the map and multimap operation is shown in the following Table.

Map	Operation
`map<Key, Element>`	A map that sorts keys with default, less<>(operator <).
`map<Key, Element, Operator>`	A map that sorts keys with Operator.
`multimap<Key, Element>`	A multimap that sorts keys with less<>(operator <).
`multimap<Key, Element, Operator>`	A multimap that sorts keys with Operator.

Table 29.1

29.2 <map> Header Members

map Operators

Operators	Description
`operator!=`	Tests if the map or multimap object on the left side of the operator is not equal to the map or multimap object on the right side.
`operator<`	Tests if the map or multimap object on the left side of the operator is less than the map or multimap object on the right side.
`operator<=`	Tests if the map or multimap object on the left side of the operator is less than or equal to the map or multimap object on the right side.
`operator==`	Tests if the map or multimap object on the left side of the operator is equal to the map or multimap object on the right side.
`operator>`	Tests if the map or multimap object on the left side of the operator is greater than the map or multimap object on the right side.
`operator>=`	Tests if the map or multimap object on the left side of the operator is greater than or equal to the map or multimap object on the right side.

Table 29.2

map Specialized Template Functions

Specialized template function	Description
`swap()`	Exchanges the elements of two maps or multimaps.

Table 29.3

map Classes

Class	Description
`value_compare Class`	Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.
`map Class`	Used for the storage and retrieval of data from a collection in which the each of the elements has a unique key with which the data is automatically ordered.
`multimap Class`	Used for the storage and retrieval of data from a collection in which the each of the elements has a key with which the data is automatically ordered and the keys do not need to have unique values.

Table 29.4

map Template Class Members

Typedefs

Template Class Member	Description
`allocator_type`	A type that represents the allocator class for the map object.
`const_iterator`	A type that provides a bidirectional iterator that can read a `const` element in the map.

const_pointer	A type that provides a pointer to a const element in a map.
const_reference	A type that provides a reference to a const element stored in a map for reading and performing const operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any const element in the map.
difference_type	A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a map.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.
key_type	A type that describes the sort key object which constitutes each element of the map.
mapped_type	A type that represents the data type stored in a map.
pointer	A type that provides a pointer to a const element in a map.
reference	A type that provides a reference to an element stored in a map.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed map.
size_type	An unsigned integer type that can represent the number of elements in a map
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the map.

Table 29.5

map Template Class Member Functions

Template class member function	Description
begin()	Returns an iterator addressing the first element in the map.
clear()	Erases all the elements of a map.
count()	Returns the number of elements in a map whose key matches a parameter-specified key.
empty()	Tests if a map is empty.
end()	Returns an iterator that addresses the location succeeding the last element in a map.
equal_range()	Returns an iterator that addresses the location succeeding the last element in a map.
erase()	Removes an element or a range of elements in a map from specified positions
find()	Returns an iterator addressing the location of an element in a map that has a key equivalent to a specified key.
get_allocator()	Returns a copy of the allocator object used to construct the map.
insert()	Inserts an element or a range of elements into the map at a specified position.
key_comp()	Retrieves a copy of the comparison object used to order keys in a map.
lower_bound()	Returns an iterator to the first element in a map that with a key value that is equal to or greater than that of a specified key.
map()	map constructor, constructs a list of a specific size or with elements of a specific value or with a specific allocator or as a copy of some other map.
max_size()	Returns the maximum length of the map.
rbegin()	Returns an iterator addressing the first element in a reversed map.
rend()	Returns an iterator that addresses the location succeeding the last element in a reversed map.
size()	Specifies a new size for a map.
swap()	Exchanges the elements of two maps.
upper_bound()	Returns an iterator to the first element in a map that with a key value that is greater than that of a specified key.
value_comp()	Retrieves a copy of the comparison object used to order element values in a map.

Table 29.6

map Template Class Operator

Operator	Description
operator[]	Inserts an element into a map with a specified key value.

Table 29.7

- The STL map class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key.
- The value of the key is unique and is used to order the data is automatically. The value of an element in a map, but not its associated key value, may be changed directly.
- Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

```
template <
   class Key,
   class Type,
   class Traits = less<Key>,
   class Allocator = allocator<pair <const Key, Type> >
>
```

Parameters

Parameter	Description
`Key`	The key data type to be stored in the map.
`Type`	The element data type to be stored in the map.
`Traits`	The type that provides a function object that can compare two element values as sort keys to determine their relative order in the map. This argument is optional and the binary predicate `less<Key>` is the default value.
`Allocator`	The type that represents the stored allocator object that encapsulates details about the map's allocation and de-allocation of memory. This argument is optional and the default value is `allocator<pair <const Key, Type> >`.

Table 29.8

- The STL map class is:
  - An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
  - Reversible, because it provides bidirectional iterators to access its elements.
  - Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
  - Unique in the sense that each of its elements must have a unique key.
  - A pair associative container, because its element data values are distinct from its key values.
  - A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

`map` Constructor

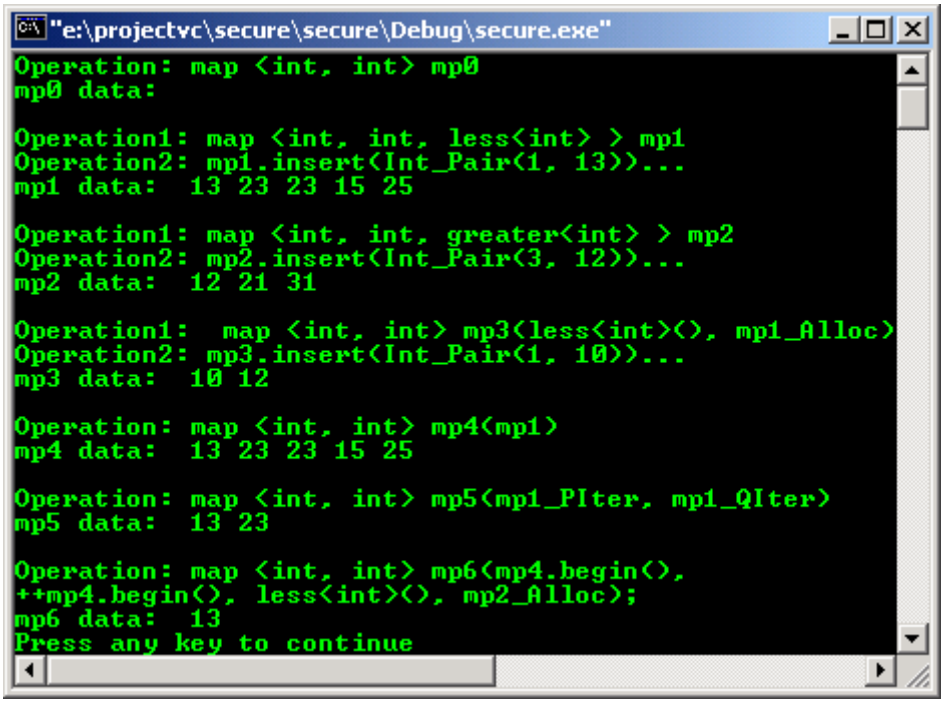
- Constructs a map that is empty or that is a copy of all or part of some other map.
- All constructors store a type of allocator object that manages memory storage for the map and that can later be returned by calling `get_allocator`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their map.
- All constructors store a function object of type Traits that is used to establish an order among the keys of the map and that can later be returned by calling `key_comp()`.
- The first three constructors specify an empty initial map, the second specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used. The key word `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the map `_Right`.
- The last three constructors copy the range `[_First, _Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

```
//map, constructor
//compiled with VC++ 7.0
//or .Net
#include <map>
#include <iostream>
using namespace std;
```

```
int main( )
{

   typedef pair<int, int> Int_Pair;
   map<int, int>::iterator mp0_Iter, mp1_Iter, mp3_Iter, mp4_Iter, mp5_Iter, mp6_Iter;
   map<int, int, greater<int> >::iterator mp2_Iter;

   //Create an empty map mp0 of key type integer
   map <int, int> mp0;

   //Create an empty map mp1 with the key comparison
   //function of less than, then insert 6 elements
   map <int, int, less<int> > mp1;
   mp1.insert(Int_Pair(1, 13));
   mp1.insert(Int_Pair(3, 23));
   mp1.insert(Int_Pair(3, 31));
   mp1.insert(Int_Pair(2, 23));
   mp1.insert(Int_Pair(6, 15));
   mp1.insert(Int_Pair(9, 25));

   //Create an empty map mp2 with the key comparison
   //function of greater than, then insert 3 elements
   map <int, int, greater<int> > mp2;
   mp2.insert(Int_Pair(3, 12));
   mp2.insert(Int_Pair(1, 31));
   mp2.insert(Int_Pair(2, 21));

   //Create a map mp3 with the
   //allocator of map mp1
   map <int, int>::allocator_type mp1_Alloc;
   mp1_Alloc = mp1.get_allocator();
   map <int, int> mp3(less<int>(), mp1_Alloc);
   mp3.insert(Int_Pair(1, 10));
   mp3.insert(Int_Pair(2, 12));

   //Create a copy, map mp4, of map mp1
   map <int, int> mp4(mp1);

   //Create a map mp5 by copying the range mp1[_First, _Last)
   map <int, int>::const_iterator mp1_PIter, mp1_QIter;
   mp1_PIter = mp1.begin();
   mp1_QIter = mp1.begin();
   mp1_QIter++;
   mp1_QIter++;
   map <int, int> mp5(mp1_PIter, mp1_QIter);

   //Create a map mp6 by copying the range mp4[_First, _Last)
   //and with the allocator of map mp2
   map <int, int>::allocator_type mp2_Alloc;
   mp2_Alloc = mp2.get_allocator();
   map <int, int> mp6(mp4.begin(), ++mp4.begin(), less<int>(), mp2_Alloc);

   //------------------------------------------------------
   cout<<"Operation: map <int, int> mp0\n";
   cout<<"mp0 data: ";
   for(mp0_Iter = mp0.begin(); mp0_Iter != mp0.end(); mp0_Iter++)
      cout<<" "<<mp0_Iter->second;
   cout<<endl;

   cout<<"\nOperation1: map <int, int, less<int> > mp1\n";
   cout<<"Operation2: mp1.insert(Int_Pair(1, 13))...\n";
   cout<<"mp1 data: ";
   for(mp1_Iter = mp1.begin(); mp1_Iter != mp1.end(); mp1_Iter++)
      cout<<" "<<mp1_Iter->second;
   cout<<endl;

   cout<<"\nOperation1: map <int, int, greater<int> > mp2\n";
   cout<<"Operation2: mp2.insert(Int_Pair(3, 12))...\n";
   cout<<"mp2 data: ";
   for(mp2_Iter = mp2.begin(); mp2_Iter != mp2.end(); mp2_Iter++)
      cout<<" "<<mp2_Iter->second;
   cout<<endl;

   cout<<"\nOperation1:  map <int, int> mp3(less<int>(), mp1_Alloc)\n";
   cout<<"Operation2: mp3.insert(Int_Pair(1, 10))...\n";
   cout<<"mp3 data: ";
   for(mp3_Iter = mp3.begin(); mp3_Iter != mp3.end(); mp3_Iter++)
      cout<<" "<<mp3_Iter->second;
   cout<<endl;
```

```
   cout<<"\nOperation: map <int, int> mp4(mp1)\n";
     cout<<"mp4 data: ";
   for(mp4_Iter = mp4.begin(); mp4_Iter != mp4.end(); mp4_Iter++)
      cout<<" "<<mp4_Iter->second;
   cout<<endl;

   cout<<"\nOperation: map <int, int> mp5(mp1_PIter, mp1_QIter)\n";
   cout<<"mp5 data: ";
   for(mp5_Iter = mp5.begin(); mp5_Iter != mp5.end(); mp5_Iter++)
      cout<<" "<<mp5_Iter->second;
   cout<<endl;

   cout<<"\nOperation: map <int, int> mp6(mp4.begin(), \n++mp4.begin(), less<int>(),
mp2_Alloc);\n";
   cout<<"mp6 data: ";
   for(mp6_Iter = mp6.begin(); mp6_Iter != mp6.end(); mp6_Iter++)
      cout<<" "<<mp6_Iter->second;
   cout<<endl;
   return 0;
}
```

Output:



----------End of map----------

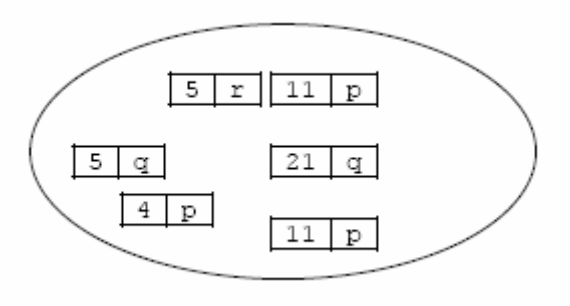
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

29.3 multimap

- A multimap is the same as **a map except that duplicates are allowed**. Thus, a multimap may contain multiple elements that have the same key. A multimap can also be used as **dictionary**.
- It can be depicted as follows:



- The iterator provided by the `map` class is a bidirectional iterator, but the class member functions `insert()` and `multimap()` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators.
- The multimap orders the sequence it controls by calling a stored function object of type `key_compare`. This stored object is a comparison function that may be accessed by calling the member function `key_comp()`.
- The `(key, value)` pairs are stored in a multimap as objects of type `pair`. The pair class requires the header <`utility`>, which is automatically included by <`map`>.

29.4 multimap Members

Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the multimap object.
`const_iterator`	A type that provides a bidirectional iterator that can read a `const` element in the multimap.
`const_pointer`	A type that provides a pointer to a `const` element in a multimap.
`const_reference`	A type that provides a reference to a `const` element stored in a multimap for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a bidirectional iterator that can read any `const` element in the multimap.
`difference_type`	A signed integer type that can be used to represent the number of elements of a multimap in a range between elements pointed to by iterators.
`iterator`	A type that provides the difference between two iterators those refer to elements within the same multimap.
`key_compare`	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the multimap.
`key_type`	A type that describes the sort key object that constitutes each element of the multimap.
`mapped_type`	A type that represents the data type stored in a multimap.
`pointer`	A type that provides a pointer to a `const` element in a multimap.
`reference`	A type that provides a reference to an element stored in a multimap.
`reverse_iterator`	A type that provides a bidirectional iterator that can read or modify an element in a reversed multimap.
`size_type`	An unsigned integer type that provides a pointer to a `const` element in a multimap
`value_type`	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the multimap

Table 29.9

Member Functions

Member function	Description
`begin()`	Returns an iterator addressing the first element in the multimap.
`clear()`	Erases all the elements of a multimap.
`count()`	Returns the number of elements in a multimap whose key matches a parameter-specified key.
`empty()`	Tests if a multimap is empty.
`end()`	Returns an iterator that addresses the location succeeding the last element in a multimap.
`equal_range()`	Returns a pair of iterators respectively to the first element in a multimap with a key that is

	greater than a specified key and to the first element in the multimap with a key that is equal to or greater than the key.
`erase()`	Removes an element or a range of elements in a multimap from specified positions or removes elements that match a specified key.
`find()`	Returns an iterator addressing the first location of an element in a multimap that has a key equivalent to a specified key.
`get_allocator()`	Returns a copy of the allocator object used to construct the multimap.
`insert()`	Inserts an element or a range of elements into a multimap.
`key_comp()`	Retrieves a copy of the comparison object used to order keys in a multimap.
`lower_bound()`	Returns an iterator to the first element in a multimap that with a key that is equal to or greater than a specified key.
`max_size()`	Returns the maximum length of the multimap.
`multimap()`	multimap constructor constructs a multimap that is empty or that is a copy of all or part of some other multimap.
`rbegin()`	Returns an iterator addressing the first element in a reversed multimap.
`rend()`	Returns an iterator that addresses the location succeeding the last element in a reversed multimap.
`size()`	Returns the number of elements in the multimap.
`swap()`	Exchanges the elements of two multimaps.
`upper_bound()`	Returns an iterator to the first element in a multimap that with a key that is greater than a specified key.
`value_comp()`	The member function returns a function object that determines the order of elements in a multimap by comparing their key values.

Table 29.10

multimap Class

- The (`key, value`) pairs are stored in a multimap as objects of type `pair`. The pair class requires the header <`utility`>, which is automatically included by <`map`>.
- The STL multimap class is used for the storage and retrieval of data from a collection in which **each element is a pair** that has both **a data value and a sort key**. The value of the key **does not need to be unique** and is **used to order the data automatically**.
- The value of an element in a multimap, but not its associated key value, **may be changed directly**. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

```
template <
   class Key,
   class Type,
   class Traits=less<Key>,
   class Allocator=allocator<pair <const Key, Type> >
>
```

Parameters

Parameter	Description
`Key`	The key data type to be stored in the multimap.
`Type`	The element data type to be stored in the multimap.
`Traits`	The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multimap. The binary predicate `less<Key>` is the default value.
`Allocator`	The type that represents the stored allocator object that encapsulates details about the map's allocation and de-allocation of memory. This argument is optional and the default value is `allocator<pair <const Key, Type> >`.

Table 29.11

- The STL multimap class is:

  - An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
  - Reversible, because it provides bidirectional iterators to access its elements.
  - Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.

- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- A pair associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

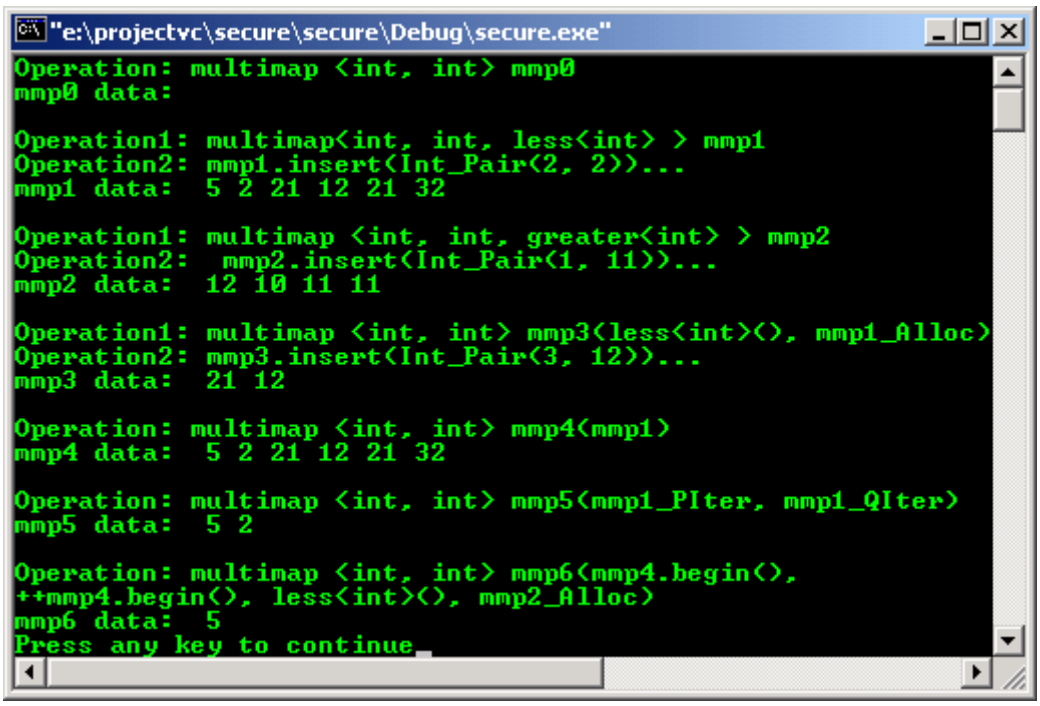
multimap Constructor

- Constructs a multimap that is empty or that is a copy of all or part of some other multimap.
- All constructors store a type of allocator object that manages memory storage for the multimap and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their multimap.
- All constructors store a function object of type `Traits` that is used to establish an order among the keys of the multimap and that can later be returned by calling `key_comp()`.
- The first three constructors specify an empty initial multimap, the second specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used. The keyword `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the multimap `_Right`.
- The last three constructors copy the range `[_First, _Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

```
//multimap, constructor or ctor
//compiled with VC++ 7.0 or .Net
//notice the duplicate key and data element
#include <map>
#include <iostream>
using namespace std;

int main()
{
   typedef pair<int, int> Int_Pair;
   multimap<int, int>::iterator mmp0Iter, mmp1Iter, mmp3Iter, mmp4Iter, mmp5Iter, mmp6Iter;
   multimap<int, int, greater<int> >::iterator mmp2Iter;

   //Create an empty multimap mmp0 of key type integer
   multimap <int, int> mmp0;

   //Create an empty multimap mmp1 with the key comparison
   //function of less than, then insert 6 elements
   multimap<int, int, less<int> > mmp1;
   mmp1.insert(Int_Pair(2, 2));
   mmp1.insert(Int_Pair(2, 21));
   mmp1.insert(Int_Pair(1, 5));
   mmp1.insert(Int_Pair(3, 12));
   mmp1.insert(Int_Pair(5, 32));
   mmp1.insert(Int_Pair(4, 21));

   //Create an empty multimap mmp2 with the key comparison
   //function of greater than, then insert 4 elements
   multimap <int, int, greater<int> > mmp2;
   mmp2.insert(Int_Pair(1, 11));
   mmp2.insert(Int_Pair(2, 10));
   mmp2.insert(Int_Pair(2, 11));
   mmp2.insert(Int_Pair(3, 12));

   //Create a multimap mmp3 with the
   //allocator of multimap mmp1
   multimap <int, int>::allocator_type mmp1_Alloc;
   mmp1_Alloc = mmp1.get_allocator();
   multimap <int, int> mmp3(less<int>(), mmp1_Alloc);
   mmp3.insert(Int_Pair(3, 12));
   mmp3.insert(Int_Pair(1, 21));

   //multimap mmp4, a copy of multimap mmp1
   multimap <int, int> mmp4(mmp1);

   //Create a multimap mmp5 by copying the range mmp1[_First, _Last)
   multimap <int, int>::const_iterator mmp1_PIter, mmp1_QIter;
   mmp1_PIter = mmp1.begin();
```

```
   mmp1_QIter = mmp1.begin();
   mmp1_QIter++;
   mmp1_QIter++;
   multimap <int, int> mmp5(mmp1_PIter, mmp1_QIter);

   //Create a multimap mmp6 by copying the range mmp4[_First, _Last)
   //and with the allocator of multimap mmp2
   multimap <int, int>::allocator_type mmp2_Alloc;
   mmp2_Alloc = mmp2.get_allocator();
   multimap <int, int> mmp6(mmp4.begin(), ++mmp4.begin(), less<int>(), mmp2_Alloc);

   //-------------------------------------------------------
   cout<<"Operation: multimap <int, int> mmp0\n";
   cout<<"mmp0 data: ";
   for(mmp0Iter = mmp0.begin(); mmp0Iter != mmp0.end(); mmp0Iter++)
      cout<<" "<<mmp0Iter->second;
   cout<<endl;

   cout<<"\nOperation1: multimap<int, int, less<int> > mmp1\n";
   cout<<"Operation2: mmp1.insert(Int_Pair(2, 2))...\n";
   cout<<"mmp1 data: ";
   for(mmp1Iter = mmp1.begin(); mmp1Iter != mmp1.end(); mmp1Iter++)
      cout<<" "<<mmp1Iter->second;
   cout<<endl;

   cout<<"\nOperation1: multimap <int, int, greater<int> > mmp2\n";
   cout<<"Operation2:  mmp2.insert(Int_Pair(1, 11))...\n";
   cout<<"mmp2 data: ";
   for(mmp2Iter = mmp2.begin(); mmp2Iter != mmp2.end(); mmp2Iter++)
      cout<<" "<<mmp2Iter->second;
   cout<<endl;

   cout<<"\nOperation1: multimap <int, int> mmp3(less<int>(), mmp1_Alloc)\n";
   cout<<"Operation2: mmp3.insert(Int_Pair(3, 12))...\n";
   cout<<"mmp3 data: ";
   for(mmp3Iter = mmp3.begin(); mmp3Iter != mmp3.end(); mmp3Iter++)
      cout<<" "<<mmp3Iter->second;
   cout<<endl;

   cout<<"\nOperation: multimap <int, int> mmp4(mmp1)\n";
   cout<<"mmp4 data: ";
   for(mmp4Iter = mmp4.begin(); mmp4Iter != mmp4.end(); mmp4Iter++)
      cout<<" "<<mmp4Iter->second;
   cout << endl;

   cout<<"\nOperation: multimap <int, int> mmp5(mmp1_PIter, mmp1_QIter)\n";
   cout<<"mmp5 data: ";
   for(mmp5Iter = mmp5.begin(); mmp5Iter != mmp5.end(); mmp5Iter++)
      cout<<" "<<mmp5Iter->second;
   cout<<endl;

   cout<<"\nOperation: multimap <int, int> mmp6(mmp4.begin(), \n++mmp4.begin(), less<int>(),
mmp2_Alloc)\n";
   cout<<"mmp6 data: ";
   for(mmp6Iter = mmp6.begin(); mmp6Iter != mmp6.end(); mmp6Iter++)
      cout<<" "<<mmp6Iter->second;
   cout<<endl;
       return 0;
}
```

Output:



----------End of multimap----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

29.5 Hash Tables

- The hash table is a data structure for collections but it is not part of the C++ standard library. It is implementation dependant.
- Libraries typically provide four kinds of hash tables that are `hash_map`, `hash_multimap`, `hash_set`, and `hash_multiset`.

29.5.1 hash_map

- The main advantage of **hashing** over **sorting** is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques.
- The value of an element in a hash_map, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.
- Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container.
- A good designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).
- This type of structure is an ordered list of uniquely occurring keywords with associated string values. If, instead, the words had more than one correct definition, so that keys were not unique, then a `hash_multimap` would be the container of choice.
- If, on the other hand, just the list of words were being stored, then a `hash_set` would be the correct container. If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure.
- The `hash_map` orders the sequence it controls by calling a stored hash `Traits` object of class `value_compare`. This stored object may be accessed by calling the member function `key_comp()`. Such a function object must behave the same as an object of class

`hash_compare<Key, less<Key> >`. Specifically, for all values `_Key` of type `Key`, the call `Traits(_Key)` yields a distribution of values of type `size_t`.

- The iterator provided by the hash_map class is a bidirectional iterator.

<hash_map> Header Members

Operators

Operator	Description
`operator!=`	Tests if the hash_map or hash_multimap object on the left side of the operator is not equal to the hash_map or hash_multimap object on the right side.
`operator<`	Tests if the hash_map or hash_multimap object on the left side of the operator is less than the hash_map or hash_multimap object on the right side.
`operator<=`	Tests if the hash_map or hash_multimap object on the left side of the operator is less than or equal to the hash_map or hash_multimap object on the right side.
`operator==`	Tests if the hash_map or hash_multimap object on the left side of the operator is equal to the hash_map or hash_multimap object on the right side.
`operator>`	Tests if the hash_map or hash_multimap object on the left side of the operator is greater than the hash_map or hash_multimap object on the right side.
`operator>=`	Tests if the hash_map or hash_multimap object on the left side of the operator is greater than or equal to the hash_map or hash_multimap object on the right side.

Table 29.12

Specialized Template Functions

Specialized template function	Description
`swap()`	Exchanges the elements of two `hash_maps` or `hash_multimaps`.

Table 29.13

Classes

Class	Description
`hash_compare Class`	Describes an object that can be used by any of the hash associative containers: hash_map, hash_multimap, hash_set, or hash_multiset, as a default `Traits` parameter object to order and hash the elements they contain.
`value_compare Class`	Provides a function object that can compare the elements of a hash_map by comparing the values of their keys to determine their relative order in the hash_map.
`hash_map Class`	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.
`hash_multimap Class`	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

Table 29.14

hash_map Template Class Members

Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the hash_map object.
`const_iterator`	A type that provides a bidirectional iterator that can read a `const` element in the hash_map.
`const_pointer`	A type that provides a pointer to a `const` element in a hash_map.
`const_reference`	A type that provides a reference to a `const` element stored in a hash_map for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a bidirectional iterator that can read any `const` element in the hash_map.
`difference_type`	A signed integer type that can be used to represent the number of elements of a hash_map in a range between elements pointed to by iterators.

iterator	A type that provides a bidirectional iterator that can read or modify any element in a hash_map.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_map**.**
key_type	A type describes the sort key object that constitutes each element of the hash_map.
mapped_type	A type that represents the data type stored in a hash_map.
pointer	A type that provides a pointer to an element in a hash_map.
reference	A type that provides a reference to an element stored in a hash_map.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_map.
size_type	An unsigned integer type that can represent the number of elements in a hash_map.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the hash_map.

table 29.15

hash_map Template Class Member Functions

Member function	Description
begin()	Returns an iterator addressing the first element in the hash_map.
clear()	Erases all the elements of a hash_map.
count()	Returns the number of elements in a hash_map whose key matches a parameter-specified key.
empty()	Tests if a hash_map is empty.
end()	Returns an iterator that addresses the location succeeding the last element in a hash_map.
equal_range()	Returns a pair of iterators, respectively, to the first element in a hash_map with a key that is greater than a specified key and to the first element in the hash_map with a key that is equal to or greater than the key.
erase()	Removes an element or a range of elements in a hash_map from specified positions
find()	Returns an iterator addressing the location of an element in a hash_map that has a key equivalent to a specified key.
get_allocator()	Returns a copy of the allocator object used to construct the hash_map.
hash_map()	Constructs a hash_map that is empty or that is a copy of all or part of some other hash_map.
insert()	Inserts an element or a range of elements into a hash_map.
key_comp()	Returns an iterator to the first element in a hash_map with a key value that is equal to or greater than that of a specified key.
lower_bound()	Returns an iterator to the first element in a hash_map with a key value that is equal to or greater than that of a specified key.
max_size()	Returns the maximum length of the hash_map.
rbegin()	Returns an iterator addressing the first element in a reversed hash_map.
rend()	Returns an iterator that addresses the location succeeding the last element in a reversed hash_map.
size()	Specifies a new size for a hash_map.
swap()	Exchanges the elements of two hash_maps.
upper_bound()	Returns an iterator to the first element in a hash_map that with a key value that is greater than that of a specified key.
value_comp()	Retrieves a copy of the comparison object used to order element values in a hash_map.

Table 29.16

hash_map Template Class Operator

Operator	Description
operator[]	Inserts an element into a hash_map with a specified key value.

Table 29.17

hash_map Class

- Stores and retrieves data quickly from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.

```
template <
   class Key,
   class Type,
   class Traits=hash_compare<Key, less<Key> >,
   class Allocator=allocator<pair <const Key, Type> >
>
```

Parameters

Parameter	Description
`Key`	The element data type to be stored in the hash_map.
`Type`	The element data type to be stored in the hash_map.
`Traits`	The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and `hash_compare<Key, less<Key> >` is the default value.
`Allocator`	The type that represents the stored allocator object that encapsulates details about the hash_map's allocation and de-allocation of memory. This argument is optional, and the default value is `allocator<pair <const Key, Type> >`.

Table 29.18

- The `hash_map` is:

  - An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
  - Reversible, because it provides a bidirectional iterator to access its elements.
  - Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
  - Unique in the sense that each of its elements must have a unique key.
  - A pair associative container, because its element data values are distinct from its key values.
  - A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

hash_map Constructor

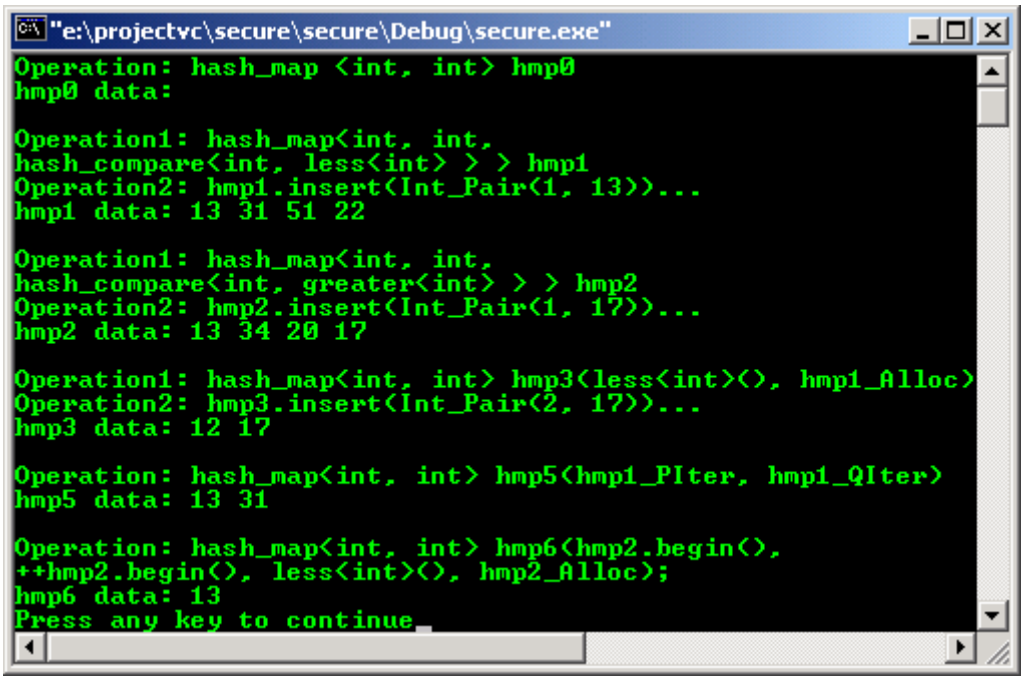
- Constructs a `hash_map` that is empty or that is a copy of all or part of some other `hash_map`.
- All constructors store a type of allocator object that manages memory storage for the hash_map and that can later be returned by calling `get_allocator`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their `hash_map`.
- All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_map` and that can later be returned by calling `key_comp`.
- The first three constructors specify an empty initial `hash_map`, the second, in addition, specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used.
- The keyword `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the `hash_map _Right`.
- The last three constructors copy the range `[_First, _Last)` of a `hash_map` with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

```
//hash_map, constructor
//compiled with visual C++ 7.0
//or VC.Net, some warnings
#include <hash_map>
#include <iostream>
using namespace std;

int main()
{
```

```
typedef pair <int, int> Int_Pair;
hash_map <int, int>::iterator hmp0_Iter, hmp1_Iter, hmp3_Iter, hmp4_Iter, hmp5_Iter, hmp6_Iter;
hash_map <int, int, hash_compare<int, greater<int> > >::iterator hmp2_Iter;

//Create an empty hash_map hmp0 of key type integer
hash_map <int, int> hmp0;

//Create an empty hash_map hmp1 with the key comparison
//function of less than, then insert 4 elements
hash_map <int, int, hash_compare <int, less<int> > > hmp1;
hmp1.insert(Int_Pair(1, 13));
hmp1.insert(Int_Pair(3, 51));
hmp1.insert(Int_Pair(7, 22));
hmp1.insert(Int_Pair(2, 31));

//Create an empty hash_map hmp2 with the key comparison
//function of greater than, then insert 4 elements
//no duplicate key...
hash_map <int, int, hash_compare <int, greater<int> > > hmp2;
hmp2.insert(Int_Pair(1, 17));
hmp2.insert(Int_Pair(2, 20));
hmp2.insert(Int_Pair(4, 13));
hmp2.insert(Int_Pair(3, 34));


//Create a hash_map hmp3 with the
//hash_map hmp1 allocator
//notice the duplicate key...
hash_map <int, int>::allocator_type hmp1_Alloc;
hmp1_Alloc = hmp1.get_allocator();
hash_map <int, int> hmp3(less<int>(), hmp1_Alloc);
hmp3.insert(Int_Pair(2, 17));
hmp3.insert(Int_Pair(1, 12));
hmp3.insert(Int_Pair(2, 15));
hmp3.insert(Int_Pair(1, 22));

//Create a hash_map hm5 by copying the range hm1[_First, _Last)
hash_map <int, int>::const_iterator hmp1_PIter, hmp1_QIter;
hmp1_PIter = hmp1.begin( );
hmp1_QIter = hmp1.begin( );
hmp1_QIter++;
hmp1_QIter++;
hash_map <int, int> hmp5(hmp1_PIter, hmp1_QIter);

//Create a hash_map hm6 by copying the range hm2[_First, _Last)
//and with the allocator of hash_map hm2
hash_map <int, int>::allocator_type hmp2_Alloc;
hmp2_Alloc = hmp2.get_allocator();
hash_map <int, int> hmp6(hmp2.begin(), ++hmp2.begin(), less<int>(), hmp2_Alloc);

//-----------------------------------
cout<<"Operation: hash_map <int, int> hmp0\n";
cout<<"hmp0 data: ";
for(hmp0_Iter = hmp0.begin(); hmp0_Iter != hmp0.end(); hmp0_Iter++)
cout<<hmp0_Iter->second<<" ";
cout<<endl;

cout<<"\nOperation1: hash_map<int, int, \nhash_compare<int, less<int> > > hmp1\n";
cout<<"Operation2: hmp1.insert(Int_Pair(1, 13))...\n";
cout<<"hmp1 data: ";
for(hmp1_Iter = hmp1.begin(); hmp1_Iter != hmp1.end(); hmp1_Iter++)
cout<<hmp1_Iter->second<<" ";
cout<<endl;

cout<<"\nOperation1: hash_map<int, int, \nhash_compare<int, greater<int> > > hmp2\n";
cout<<"Operation2: hmp2.insert(Int_Pair(1, 17))...\n";
cout<<"hmp2 data: ";
for(hmp2_Iter = hmp2.begin(); hmp2_Iter != hmp2.end(); hmp2_Iter++)
cout<<hmp2_Iter->second<<" ";
cout<<endl;

cout<<"\nOperation1: hash_map<int, int> hmp3(less<int>(), hmp1_Alloc)\n";
cout<<"Operation2: hmp3.insert(Int_Pair(2, 17))...\n";
cout<<"hmp3 data: ";
for(hmp3_Iter = hmp3.begin(); hmp3_Iter != hmp3.end(); hmp3_Iter++)
cout<<hmp3_Iter->second<<" ";
cout<<endl;

cout<<"\nOperation: hash_map<int, int> hmp5(hmp1_PIter, hmp1_QIter)\n";
cout<<"hmp5 data: ";
```

```
for(hmp5_Iter = hmp5.begin(); hmp5_Iter != hmp5.end(); hmp5_Iter++)
cout<<hmp5_Iter->second<<" ";
cout<<endl;

cout<<"\nOperation: hash_map<int, int> hmp6(hmp2.begin(), \n++hmp2.begin(), less<int>(),
hmp2_Alloc);\n";
cout<<"hmp6 data: ";
for(hmp6_Iter = hmp6.begin(); hmp6_Iter != hmp6.end(); hmp6_Iter++)
cout<<hmp6_Iter->second <<" ";
cout<<endl;
return 0;
}
```

Output:



----------End of hash_map----------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

29.5.2 hash_multimap Members

Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the hash_multimap object.
`const_iterator`	A type that provides a bidirectional iterator that can read a `const` element in the hash_multimap.
`const_pointer`	A type that provides a pointer to a `const` element in a hash_multimap.
`const_reference`	A type that provides a reference to a `const` element stored in a hash_multimap for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a bidirectional iterator that can read any `const` element in the hash_multimap.
`difference_type`	A signed integer type that can be used to represent the number of elements of a hash_multimap in a range between elements pointed to by iterators.
`iterator`	A type that provides a bidirectional iterator that can read or modify any element in a hash_multimap.
`key_compare`	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_multimap.
`key_type`	A type that describes the sort key object that constitutes each element of the hash_multimap.

mapped_type	A type that represents the data type stored in a hash_multimap.
pointer	A type that provides a pointer to an element in a hash_multimap.
reference	A type that provides a reference to an element stored in a hash_multimap.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multimap.
size_type	An unsigned integer type that can represent the number of elements in a hash_multimap.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the hash_multimap.

Table 29.19

Member Functions

Member function	Description
begin()	Returns an iterator addressing the first element in the hash_multimap.
clear()	Erases all the elements of a hash_multimap.
count()	Returns the number of elements in a hash_multimap whose key matches a parameter-specified key.
empty()	Tests if a hash_multimap is empty.
end()	Returns an iterator that addresses the location succeeding the last element in a hash_multimap.
equal_range()	Returns an iterator that addresses the location succeeding the last element in a hash_multimap.
erase()	Removes an element or a range of elements in a hash_multimap from specified positions
find()	Returns an iterator addressing the location of an element in a hash_multimap that has a key equivalent to a specified key.
get_allocator()	Returns a copy of the allocator object used to construct the hash_multimap.
hash_multimap()	hash_multimap constructor, constructs a list of a specific size or with elements of a specific value or with a specific allocator or as a copy of some other hash_multimap.
insert()	Inserts an element or a range of elements into the hash_multimap at a specified position.
key_comp()	Retrieves a copy of the comparison object used to order keys in a hash_multimap.
lower_bound()	Returns an iterator to the first element in a hash_multimap that with a key value that is equal to or greater than that of a specified key.
max_size()	Returns the maximum length of the hash_multimap.
rbegin()	Returns an iterator addressing the first element in a reversed hash_multimap.
rend()	Returns an iterator that addresses the location succeeding the last element in a reversed hash_multimap.
size()	Specifies a new size for a hash_multimap.
swap()	Exchanges the elements of two hash_multimaps.
upper_bound()	Returns an iterator to the first element in a hash_multimap that with a key value that is greater than that of a specified key.
value_comp()	Retrieves a copy of the comparison object used to order element values in a hash_multimap.

Table 29.20

- The container class `hash_multimap` is an extension of the STL and is used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

```
template <
   class Key,
   class Type,
   class Traits = hash_compare<Key, less<Key> >,
   class Allocator = allocator<pair <const Key, Type> >
>
```

Parameters

Parameter	Description
Key	The element data type to be stored in the hash_multimap.
Type	The element data type to be stored in the hash_multimap.
Traits	The type that includes two function objects, one of class `Traits` that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.

`Allocator`	The type that represents the stored allocator object that encapsulates details about the hash_multimap's allocation and de-allocation of memory. This argument is optional, and the default value is `allocator<pair <const Key, Type> >`.

Table 29.21

- The hash_multimap is:

  - An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
  - Reversible, because it provides a bidirectional iterator to access its elements.
  - Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
  - Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
  - A pair associative container, because its element values are distinct from its key values.
  - A template class, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

- The `hash_multimap` orders the sequence it controls by calling a stored hash `Traits` object of type `value_compare()`. This stored object may be accessed by calling the member function `key_comp()`.
- Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values `_Key` of type `Key`, the call `Traits(_Key)` yields a distribution of values of type `size_t`.
- The iterator provided by the hash_multimap class is a bidirectional iterator, but the class member functions `insert()` and `hash_multimap()` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators.

hash_multimap Constructor

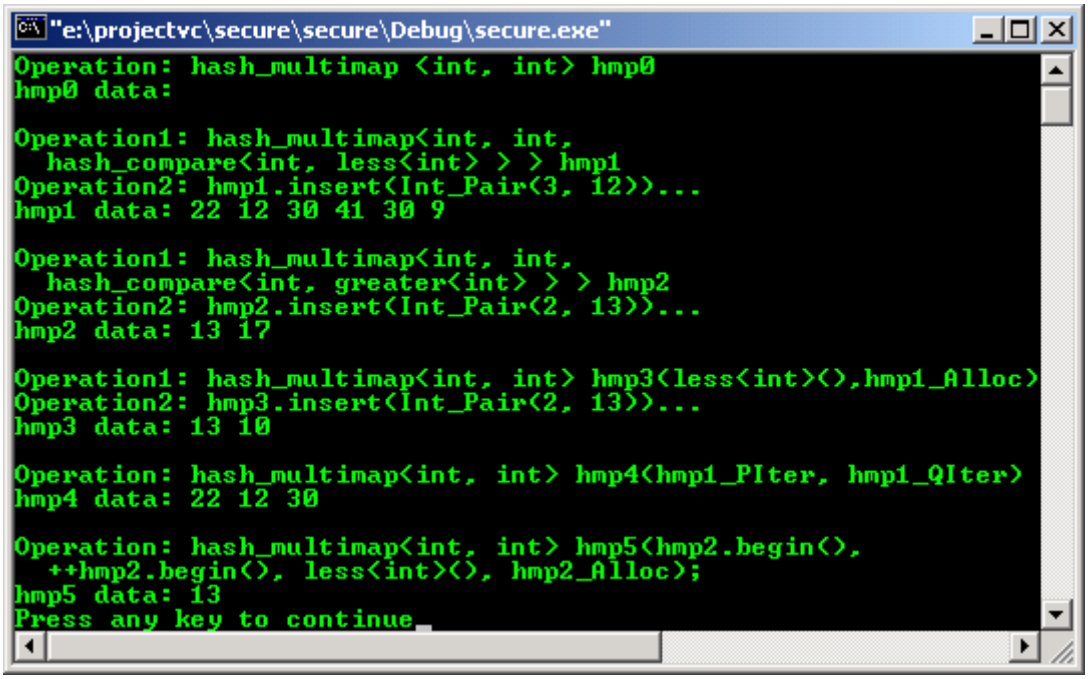
- Constructs a `hash_multimap` that is empty or that is a copy of all or part of some other `hash_multimap`.
- All constructors store a type of allocator object that manages memory storage for the hash_multimap and that can later be returned by calling `get_allocator()`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their `hash_multimap`.
- All constructors store a function object of type `Traits` that is used to establish an order among the keys of the hash_multimap and that can later be returned by calling `key_comp()`.
- The first three constructors specify an empty initial `hash_multimap`, the second specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used. The keyword `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the `hash_multimap _Right`.
- The last three constructors copy the range `[_First, _Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

```
//hash_multimap, constructor
//compiled with VC7.0 or .Net
//a lot of warning messages:-)
#include <hash_map>
#include <iostream>
using namespace std;

int main()
{
   typedef pair <int, int> Int_Pair;
   hash_multimap <int, int>::iterator hmp0_Iter, hmp1_Iter, hmp3_Iter, hmp4_Iter, hmp5_Iter;
   hash_multimap <int, int, hash_compare <int, greater<int> > >::iterator hmp2_Iter;

   //Create an empty hash_multimap hmp0 of key type integer
   hash_multimap <int, int> hmp0;
```

```
   //Create an empty hash_multimap hmp1 with the key comparison
   //function of less than, then insert 6 elements
   hash_multimap <int, int, hash_compare <int, less<int> > > hmp1;
   hmp1.insert(Int_Pair(3, 12));
   hmp1.insert(Int_Pair(2, 30));
   hmp1.insert(Int_Pair(1, 22));
   hmp1.insert(Int_Pair(7, 41));
   hmp1.insert(Int_Pair(4, 9));
   hmp1.insert(Int_Pair(7, 30));

   //Create an empty hash_multimap hmp2 with the key comparison
   //function of greater than, then insert 2 elements
   hash_multimap <int, int, hash_compare <int, greater<int> > > hmp2;
   hmp2.insert(Int_Pair(2, 13));
   hmp2.insert(Int_Pair(1, 17));

   //Create a hash_multimap hmp3 with the
   //allocator of hash_multimap hmp1
   hash_multimap <int, int>::allocator_type hmp1_Alloc;
   hmp1_Alloc = hmp1.get_allocator();
   hash_multimap <int, int> hmp3(less<int>(),hmp1_Alloc);
   hmp3.insert(Int_Pair(2, 13));
   hmp3.insert(Int_Pair(4, 10));

   //Create a hash_multimap hmp4 by copying the range hmp1[_First, _Last)
   hash_multimap <int, int>::const_iterator hmp1_PIter, hmp1_QIter;
   hmp1_PIter = hmp1.begin();
   hmp1_QIter = hmp1.begin();
   hmp1_QIter++;
   hmp1_QIter++;
   hmp1_QIter++;
   hash_multimap <int, int> hmp4(hmp1_PIter, hmp1_QIter);

   //Create a hash_multimap hmp5 by copying the range hmp2[_First, _Last)
   //and with the allocator of hash_multimap hmp2
   hash_multimap <int, int>::allocator_type hmp2_Alloc;
   hmp2_Alloc = hmp2.get_allocator();
   hash_multimap <int, int> hmp5(hmp2.begin(), ++hmp2.begin(), less<int>(), hmp2_Alloc);

   //-------------------------------------------------
   cout<<"Operation: hash_multimap <int, int> hmp0\n";
   cout<<"hmp0 data: ";
   for(hmp0_Iter = hmp0.begin(); hmp0_Iter != hmp0.end(); hmp0_Iter++)
      cout<<hmp0_Iter->second<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multimap<int, int, \n  hash_compare<int, less<int> > > hmp1\n";
   cout<<"Operation2: hmp1.insert(Int_Pair(3, 12))...\n";
   cout<<"hmp1 data: ";
   for(hmp1_Iter = hmp1.begin(); hmp1_Iter != hmp1.end(); hmp1_Iter++)
      cout<<hmp1_Iter->second<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multimap<int, int, \n  hash_compare<int, greater<int> > > hmp2\n";
    cout<<"Operation2: hmp2.insert(Int_Pair(2, 13))...\n";
   cout<<"hmp2 data: ";
   for(hmp2_Iter = hmp2.begin(); hmp2_Iter != hmp2.end(); hmp2_Iter++)
   cout<<hmp2_Iter->second<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multimap<int, int> hmp3(less<int>(), hmp1_Alloc)\n";
   cout<<"Operation2: hmp3.insert(Int_Pair(2, 13))...\n";
   cout<<"hmp3 data: ";
   for(hmp3_Iter = hmp3.begin(); hmp3_Iter != hmp3.end(); hmp3_Iter++)
      cout<<hmp3_Iter->second<<" ";
   cout<<endl;

   cout<<"\nOperation: hash_multimap<int, int> hmp4(hmp1_PIter, hmp1_QIter)\n";
   cout<<"hmp4 data: ";
   for(hmp4_Iter = hmp4.begin(); hmp4_Iter != hmp4.end(); hmp4_Iter++)
      cout<<hmp4_Iter->second<<" ";
   cout<<endl;

   cout<<"\nOperation: hash_multimap<int, int> hmp5(hmp2.begin(), \n  ++hmp2.begin(),
less<int>(), hmp2_Alloc);\n";
   cout<<"hmp5 data: ";
   for(hmp5_Iter = hmp5.begin(); hmp5_Iter != hmp5.end(); hmp5_Iter++)
      cout<<hmp5_Iter->second<<" ";
   cout<<endl;
```

```
    return 0;
}
```

Output:



----------End of hash_multimap----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

29.5.3 hash_set

- The elements of a `hash_set` are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once.
- If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure. If unique definitions were attached as values to the list of key words, then a `hash_map` would be an appropriate structure to contain this data. If instead the definitions were not unique, then a `hash_multimap` would be the container of choice.
- The `hash_set` orders the sequence it controls by calling a stored hash `Traits` object of type `value_compare`.
- This stored object may be accessed by calling the member function `key_comp()`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values `_Key` of type `Key`, the call `Trait(_Key)` yields a distribution of values of type `size_t`.
- The iterator provided by the `hash_set` class is a bidirectional iterator.

<hash_set> Header Members

Operators

Operator	Description
`operator!=`	Tests if the hash_set or hash_multiset object on the left side of the operator is not equal to the hash_set or hash_multiset object on the right side.
`operator<`	Tests if the hash_set or hash_multiset object on the left side of the operator is less than the hash_set or hash_multiset object on the right side.
`operator<=`	Tests if the hash_set or hash_multiset object on the left side of the operator is less than or equal to the hash_set or hash_multiset object on the right side.
`operator==`	Tests if the hash_set or hash_multiset object on the left side of the operator is equal to the

	hash_set or hash_multiset object on the right side.
operator>	Tests if the hash_set or hash_multiset object on the left side of the operator is greater than the hash_set or hash_multiset object on the right side.
operator>=	Tests if the hash_set or hash_multiset object on the left side of the operator is greater than or equal to the hash_set or hash_multiset object on the right side.

Table 29.22

Specialized Template Functions

Specialized template function	Description
swap()	Exchanges the elements of two hash_sets or hash_multisets.

Table 29.23

Classes

Class	Description
hash_compare Class	Describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default Traits parameter object to order and hash the elements they contain.
hash_set Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.
hash_multiset Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

Table 29.24

hash_set Template Class Members

Typedefs

Typedef	Description
allocator_type	A type that represents the allocator class for the hash_set object.
const_iterator	A type that provides a bidirectional iterator that can read a const element in the hash_set.
const_pointer	A type that provides a pointer to a const element in a hash_set.
const_reference	A type that provides a reference to a const element stored in a hash_set for reading and performing const operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any const element in the hash_set.
difference_type	A signed integer type that can be used to represent the number of elements of a hash_set in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a hash_set.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_set.
key_type	A type that describes an object stored as an element of a hash_set in its capacity as sort key.
pointer	A type that provides a pointer to an element in a hash_set.
reference	A type that provides a reference to an element stored in a hash_set
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_set.
size_type	An unsigned integer type that can represent the number of elements in a hash_set.
value_compare	A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_set to determine their relative order and a unary predicate that hashes the elements.
value_type	A type that describes an object stored as an element of a hash_set in its capacity as a value.

Table 29.25

hash_set Template Class Member Functions

Member function	Description
`begin()`	Returns an iterator that addresses the first element in the hash_set.
`clear()`	Erases all the elements of a hash_set.
`count()`	Returns the number of elements in a hash_set whose key matches a parameter-specified key.
`empty()`	Tests if a hash_set is empty.
`end()`	Returns an iterator that addresses the location succeeding the last element in a hash_set.
`equal_range()`	Returns a pair of iterators respectively to the first element in a hash_set with a key that is greater than a specified key and to the first element in the hash_set with a key that is equal to or greater than the key.
`erase()`	Removes an element or a range of elements in a hash_set from specified positions or removes elements that match a specified key.
`find()`	Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.
`get_allocator()`	Returns a copy of the allocator object used to construct the hash_set.
`hash_set()`	Constructs a hash_set that is empty or that is a copy of all or part of some other hash_set.
`insert()`	Inserts an element or a range of elements into a hash_set.
`key_comp()`	Retrieves a copy of the comparison object used to order keys in a hash_set.
`lower_bound()`	Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.
`max_size()`	Returns the maximum length of the hash_set.
`rbegin()`	Returns an iterator addressing the first element in a reversed hash_set.
`rend()`	Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.
`size()`	Returns the number of elements in the hash_set.
`swap()`	Exchanges the elements of two hash_sets.
`upper_bound()`	Returns an iterator to the first element in a hash_set that with a key that is equal to or greater than a specified key.
`value_comp()`	Retrieves a copy of the hash traits object used to hash and order element key values in a hash_set.

Table 29.26

hash_set Class

- The container class hash_set is an extension of the Standard Template Library (STL) and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

```
template <
   class Key,
   class Traits=hash_compare<Key, less<Key> >,
   class Allocator=allocator<Key>
>
```

Parameters

Parameter	Description
`Key`	The element data type to be stored in the hash_set.
`Traits`	The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.
`Allocator`	The type that represents the stored allocator object that encapsulates details about the hash_set's allocation and de-allocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Table 29.27

- The `hash_set` is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because hash_set is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

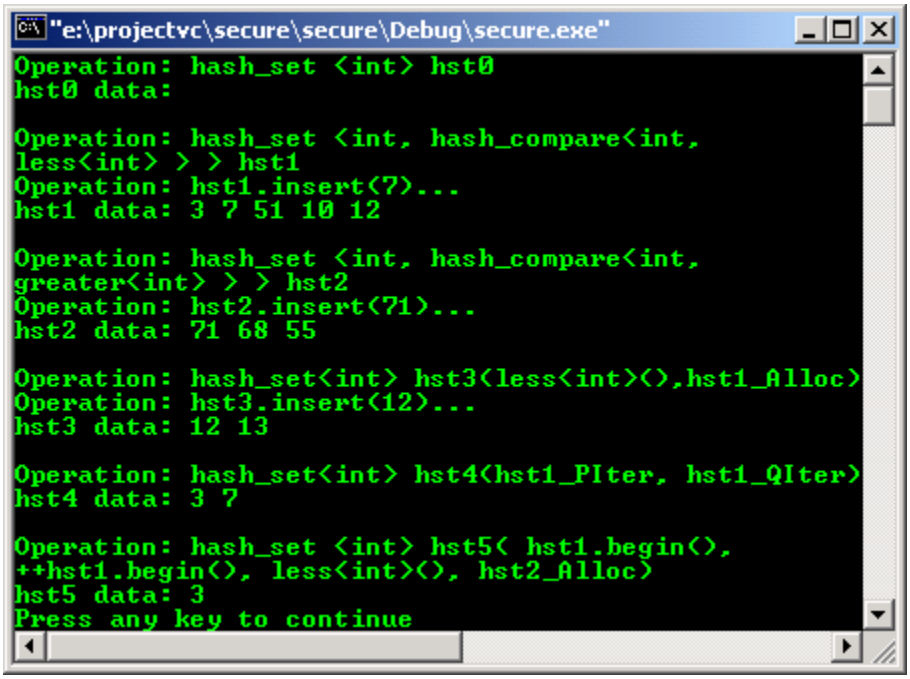
hash_set Constructor

- Constructs a `hash_set` that is empty or that is a copy of all or part of some other `hash_set`.
- All constructors store a type of allocator object that manages memory storage for the `hash_set` and that can later be returned by calling `get_allocator()`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their hash_sets.
- All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_set` and that can later be returned by calling `key_comp`.
- The first three constructors specify an empty initial `hash_set`, the second specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used.
- The key word `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the `hash_set _Right`.
- The last three constructors copy the range `[_First, _Last)` of a `hash_set` with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.
- The actual order of elements in a `hash_set` container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

```
//hash_set, constructor
//compiled with VC7.0/.Net
//some warnings
#include <hash_set>
#include <iostream>
using namespace std;

int main()
{
hash_set <int>::iterator hst0_Iter, hst1_Iter, hst3_Iter, hst4_Iter, hst5_Iter;
hash_set <int, hash_compare <int, greater<int> > >::iterator hst2_Iter;

//Create an empty hash_set hst0 of key type integer
hash_set <int> hst0;

//Create an empty hash_set hst1 with the key comparison
//function of less than, then insert 5 elements
hash_set <int, hash_compare<int, less<int> > > hst1;
hst1.insert(7);
hst1.insert(3);
hst1.insert(12);
hst1.insert(51);
hst1.insert(10);

//Create an empty hash_set hst2 with the key comparison
//function of greater than, then insert 4 elements
hash_set<int, hash_compare<int, greater<int> > > hst2;
hst2.insert(71);
hst2.insert(68);
hst2.insert(68);
hst2.insert(55);

//Create a hash_set hst3 with the
//hash_set hst1 allocator
hash_set<int>::allocator_type hst1_Alloc;
hst1_Alloc = hst1.get_allocator();
hash_set<int> hst3(less<int>(),hst1_Alloc);
```

```
hst3.insert(12);
hst3.insert(13);
hst3.insert(12);

//Create a hash_set hst4 by copying the range hst1[_First, _Last)
hash_set <int>::const_iterator hst1_PIter, hst1_QIter;
hst1_PIter = hst1.begin();
hst1_QIter = hst1.begin();
hst1_QIter++;
hst1_QIter++;
hash_set<int> hst4(hst1_PIter, hst1_QIter);

//Create a hash_set hst5 by copying the range hst4[_First, _Last)
//and with the allocator of hash_set hst2
hash_set <int>::allocator_type hst2_Alloc;
hst2_Alloc = hst2.get_allocator();
hash_set <int> hst5(hst1.begin(), ++hst1.begin(), less<int>(), hst2_Alloc);

//------------------------------------------------
cout<<"Operation: hash_set <int> hst0\n";
cout<<"hst0 data: ";
for(hst0_Iter = hst0.begin(); hst0_Iter != hst0.end(); hst0_Iter++)
cout<<*hst0_Iter<<" ";
cout<<endl;

cout<<"\nOperation: hash_set <int, hash_compare<int, \nless<int> > > hst1\n";
cout<<"Operation: hst1.insert(7)...\n";
cout<< "hst1 data: ";
for(hst1_Iter = hst1.begin(); hst1_Iter != hst1.end(); hst1_Iter++)
cout<<*hst1_Iter << " ";
cout<<endl;

cout<<"\nOperation: hash_set <int, hash_compare<int, \ngreater<int> > > hst2\n";
cout<<"Operation: hst2.insert(71)...\n";
cout<<"hst2 data: ";
for(hst2_Iter = hst2.begin(); hst2_Iter != hst2.end(); hst2_Iter++)
cout<<*hst2_Iter<<" ";
cout<<endl;

cout<<"\nOperation: hash_set<int> hst3(less<int>(),hst1_Alloc)\n";
cout<<"Operation: hst3.insert(12)...\n";
cout<<"hst3 data: ";
for(hst3_Iter = hst3.begin(); hst3_Iter != hst3.end(); hst3_Iter++)
cout<<*hst3_Iter<<" ";
cout<<endl;

cout<<"\nOperation: hash_set<int> hst4(hst1_PIter, hst1_QIter)\n";
cout<<"hst4 data: ";
for(hst4_Iter = hst4.begin(); hst4_Iter != hst4.end(); hst4_Iter++)
cout<<*hst4_Iter<<" ";
cout<<endl;

cout<<"\nOperation: hash_set <int> hst5(hst1.begin(), \n++hst1.begin(), less<int>(),
hst2_Alloc)\n";
cout<<"hst5 data: ";
for(hst5_Iter = hst5.begin(); hst5_Iter != hst5.end(); hst5_Iter++)
cout<<*hst5_Iter<<" ";
cout<<endl;
return 0;
}
```

Output:



----------End of the hash_set----------
---www.tenouk.com---

1. Check the best selling C / C++ and STL books at Amazon.com.

29.5.4 hash_multiset Members

Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the hash_multiset object.
`const_iterator`	A type that provides a bidirectional iterator that can read a `const` element in the hash_multiset.
`const_pointer`	A type that provides a pointer to a `const` element in a hash_multiset.
`const_reference`	A type that provides a reference to a `const` element stored in a hash_multiset for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a bidirectional iterator that can read any `const` element in the hash_multiset.
`difference_type`	A signed integer type that provides the difference between two iterators that address elements within the same hash_multiset.
`iterator`	A type that provides a bidirectional iterator that can read or modify any element in a hash_multiset.
`key_compare`	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_multiset.
`key_type`	A type that provides a function object that can compare sort keys to determine the relative order of two elements in the hash_multiset.
`pointer`	A type that provides a pointer to an element in a hash_multiset
`reference`	A type that provides a reference to an element stored in a hash_multiset.
`reverse_iterator`	A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multiset.
`size_type`	An unsigned integer type that can represent the number of elements in a hash_multiset.
`value_compare`	A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.
`value_type`	A type that describes an object stored as an element of a hash_multiset in its capacity as a value.

Table 29.28

Member Functions

Member function	Description
`begin()`	Returns an iterator that addresses the first element in the hash_multiset.
`clear()`	Erases all the elements of a hash_multiset.
`count()`	Returns the number of elements in a hash_multiset whose key matches a parameter-specified key
`empty()`	Tests if a hash_multiset is empty.
`end()`	Returns an iterator that addresses the location succeeding the last element in a hash_multiset.
`equal_range()`	Returns a pair of iterators respectively to the first element in a hash_multiset with a key that is greater than a specified key and to the first element in the hash_multiset with a key that is equal to or greater than the key.
`erase()`	Removes an element or a range of elements in a hash_multiset from specified positions or removes elements that match a specified key.
`find()`	Returns an iterator addressing the location of an element in a hash_multiset that has a key equivalent to a specified key.
`get_allocator()`	Returns a copy of the allocator object used to construct the hash_multiset.
`hash_multiset()`	hash_multiset constructor, constructs a hash_multiset that is empty or that is a copy of all or part of some other hash_multiset.
`insert()`	Inserts an element or a range of elements into a hash_multiset.
`key_comp()`	Retrieves a copy of the comparison object used to order keys in a hash_multiset.
`lower_bound()`	Returns an iterator to the first element in a hash_multiset with a key that is equal to or greater than a specified key.
`max_size()`	Returns the maximum length of the hash_multiset.
`rbegin()`	Returns an iterator addressing the first element in a reversed hash_multiset.
`rend()`	Returns an iterator that addresses the location succeeding the last element in a reversed hash_multiset.
`size()`	Returns the number of elements in the hash_multiset.
`swap()`	Exchanges the elements of two hash_multisets.
`upper_bound()`	Returns an iterator to the first element in a hash_multiset that with a key that is equal to or greater than a specified key.
`value_comp()`	Retrieves a copy of the hash traits object used to hash and order element key values in a hash_multiset.

Table 29.29

hash_multiset Class

- The container class hash_multiset is an extension of the Standard Template Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained serve as the key values and are not required to be unique.

```
template <
   class Key,
   class Traits = hash_compare<Key, less<Key> >,
   class Allocator = allocator<Key>
>
```

Parameters

Parameter	Description
`Key`	The element data type to be stored in the hash_multiset.
`Traits`	The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.
`Allocator`	The type that represents the stored allocator object that encapsulates details about the hash_multiset's allocation and de-allocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Table 29.30

- The `hash_multiset` is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because `hash_multiset` is also a simple associative container, its elements are also unique.
- A template class because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

- The elements of a `hash_multiset` may be multiple and serve as their own sort keys, so keys are not unique.
- The `hash_multiset` orders the sequence it controls by calling a stored hash traits object of type `value_compare`. This stored object may be accessed by calling the member function `key_comp()`. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values `Key` of type `Key`, the call `Trait(Key)` yields a distribution of values of type `size_t`.
- Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.
- The iterator provided by the `hash_multiset` class is a bidirectional iterator, but the class member functions `insert()` and `hash_multiset()` have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators.

hash_multiset Constructor

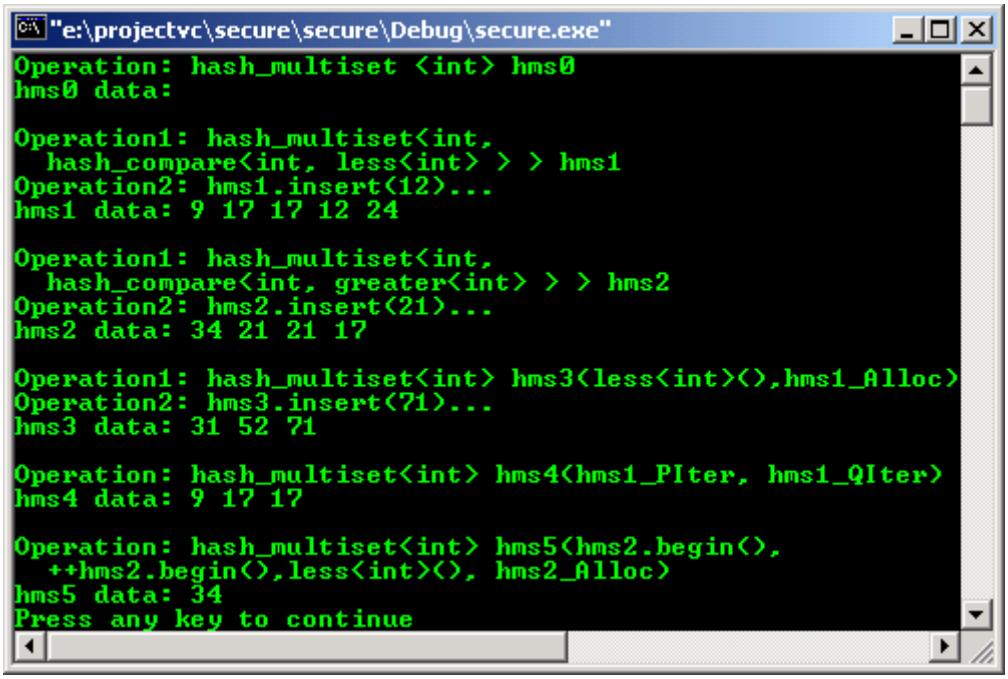
- Constructs a `hash_multiset` that is empty or that is a copy of all or part of some other `hash_multiset`.
- All constructors store a type of allocator object that manages memory storage for the `hash_multiset` and that can later be returned by calling `get_allocator()`.
- The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors initialize their `hash_multisets`.
- All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_multiset` and that can later be returned by calling `key_comp()`.
- The first three constructors specify an empty initial `hash_multiset`, the second specifying the type of comparison function `(_Comp)` to be used in establishing the order of the elements and the third explicitly specifying the allocator type `(_Al)` to be used. The keyword `explicit` suppresses certain kinds of automatic type conversion.
- The fourth constructor specifies a copy of the `hash_multiset _Right`.
- The last three constructors copy the range `[_First, _Last)` of a `hash_multiset` with increasing explicitness in specifying the type of comparison function of class Compare and allocator.
- The actual order of elements in a `hash_set` container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

```
//hash_multiset, constructor
//compiled with VC7.0 or .Net
//a lot of warning messages...
#include <hash_set>
#include <iostream>
using namespace std;

int main()
{
   hash_multiset <int>::iterator hms0_Iter, hms1_Iter, hms3_Iter, hms4_Iter, hms5_Iter;
   hash_multiset <int, hash_compare <int, greater<int> > >::iterator hms2_Iter;

   //Create an empty hash_multiset hms0 of key type integer
   hash_multiset <int> hms0;

   //Create an empty hash_multiset hms1 with the key comparison
```

```
   //function of less than, then insert 6 elements
   hash_multiset<int, hash_compare<int, less<int> > > hms1;
   hms1.insert(12);
   hms1.insert(17);
   hms1.insert(24);
   hms1.insert(17);
   hms1.insert(9);

   //Create an empty hash_multiset hms2 with the key comparison
   //function of greater than, then insert 4 elements
   hash_multiset<int, hash_compare<int, greater<int> > > hms2;
   hms2.insert(21);
   hms2.insert(34);
   hms2.insert(21);
   hms2.insert(17);

   //Create a hash_multiset hms3 with the
   //allocator of hash_multiset hms1
   hash_multiset <int>::allocator_type hms1_Alloc;
   hms1_Alloc = hms1.get_allocator();
   hash_multiset <int> hms3(less<int>(), hms1_Alloc);
   hms3.insert(71);
   hms3.insert(52);
   hms3.insert(31);

   //Create a hash_multiset hms4 by copying the range hms1[_First, _Last)
   hash_multiset <int>::const_iterator hms1_PIter, hms1_QIter;
   hms1_PIter = hms1.begin();
   hms1_QIter = hms1.begin();
   hms1_QIter++;
   hms1_QIter++;
   hms1_QIter++;
   hash_multiset<int> hms4(hms1_PIter, hms1_QIter);

   //Create a hash_multiset hms5 by copying the range hms2[_First, _Last)
   //and with the allocator of hash_multiset hms2
   hash_multiset<int>::allocator_type hms2_Alloc;
   hms2_Alloc = hms2.get_allocator( );
   hash_multiset<int> hms5(hms2.begin(), ++hms2.begin(),less<int>(), hms2_Alloc);

   //-------------------------------------------------------
   cout<<"Operation: hash_multiset <int> hms0\n";
   cout<<"hms0 data: ";
   for(hms0_Iter = hms0.begin(); hms0_Iter != hms0.end(); hms0_Iter++)
      cout<<*hms0_Iter<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multiset<int, \n  hash_compare<int, less<int> > > hms1\n";
   cout<<"Operation2: hms1.insert(12)...\n";
   cout<<"hms1 data: ";
   for(hms1_Iter = hms1.begin(); hms1_Iter != hms1.end(); hms1_Iter++)
      cout<<*hms1_Iter<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multiset<int, \n  hash_compare<int, greater<int> > > hms2\n";
   cout<<"Operation2: hms2.insert(21)...\n";
   cout<<"hms2 data: ";
   for(hms2_Iter = hms2.begin(); hms2_Iter != hms2.end(); hms2_Iter++)
      cout<<*hms2_Iter<<" ";
   cout<<endl;

   cout<<"\nOperation1: hash_multiset<int> hms3(less<int>(),hms1_Alloc)\n";
   cout<<"Operation2: hms3.insert(71)...\n";
   cout<<"hms3 data: ";
   for(hms3_Iter = hms3.begin(); hms3_Iter != hms3.end(); hms3_Iter++)
      cout<<*hms3_Iter<<" ";
   cout<<endl;

   cout<<"\nOperation: hash_multiset<int> hms4(hms1_PIter, hms1_QIter)\n";
   cout<<"hms4 data: ";
   for(hms4_Iter = hms4.begin(); hms4_Iter != hms4.end(); hms4_Iter++)
      cout<<*hms4_Iter<<" ";
   cout<<endl;

   cout<<"\nOperation: hash_multiset<int> hms5(hms2.begin(), \n ++hms2.begin(), less<int>(),
hms2_Alloc)\n";
   cout<<"hms5 data: ";
   for(hms5_Iter = hms5.begin(); hms5_Iter != hms5.end(); hms5_Iter++)
      cout<<*hms5_Iter<<" ";
   cout<<endl;
```

```
    return 0;
}
```

Output:



29.6 Strings

- You can also use strings as STL containers. By *strings* that mean objects of the C++ string classes, `basic_string<>`, `string`, and `wstring`. Strings are similar to vectors except that their elements are characters. This has been discussed extensively in Module 25 and 26.

29.7 Ordinary Arrays

- An ordinary C and C++ language array type that has static or dynamic size is a container. However, ordinary arrays are not STL containers because they don't provide member functions such as `size()` and `empty()`.
- However, the STL's design allows you to call algorithms for these ordinary arrays. This is especially useful when you process static arrays of values as an initializer list.
- You should have familiar with this traditional array, what is new in STL is using algorithms for them.
- Note that in C++ it is no longer necessary to program dynamic arrays directly. Vectors provide all properties of dynamic arrays with a safer and more convenient interface.

29.8 Some Summary

No	Sequences container	Summary
1	`vector`	A sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a `vector` may vary dynamically; memory management is automatic. `vector` is the simplest of the STL container classes, and in many cases the most efficient.
2	`deque`	Like a `vector` with extra features that `deque` does not have any member functions analogous to `vector`'s `capacity()` and `reserve()`, and does not provide any of the guarantees on iterator validity that are associated with those member functions. The Standard Template Library (STL) sequence container deque arranges elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container. However, unlike a vector, the deque class also supports efficient insertion and deletion at the front of the container.
4	`list`	A doubly linked list. It is a sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at

		the beginning or the end, or in the middle. `List`s have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, `list<Type>::iterator` might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.
	Associative container	**Summary**
6	`set`	A **sorted associative container** that stores objects of type `Key`. `Set` is a simple associative container, meaning that its value type, as well as its key type, is `Key`. It is also a unique associative container, meaning that no two elements are the same. The set algorithms require their arguments to be sorted ranges, and, since `set` and `multiset` are sorted associative containers, their elements are always sorted in ascending order. The output range of these algorithms is always sorted, and inserting a sorted range into a `set` or `multiset` is a fast operation: the **unique sorted associative container** and multiple **sorted associative container** requirements guarantee that inserting a range takes only linear time if the range is already sorted. `Set` has the important property that inserting a new element into a `set` does not invalidate iterators that point to existing elements. Erasing an element from a set also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.
7	`multiset`	`Multiset` is a sorted associative container that stores objects of type `Key`. `Multiset` is a simple associative container, meaning that its value type, as well as its key type, is `Key`. It is also a multiple associative container, meaning that two or more elements may be identical. The set algorithms require their arguments to be sorted ranges, and, since `set` and `multiset` are sorted associative containers, their elements are always sorted in ascending order. The output range of these algorithms is always sorted, and inserting a sorted range into a `set` or `multiset` is a fast operation: the **unique sorted associative container** and **multiple sorted associative container** requirements guarantee that inserting a range takes only linear time if the range is already sorted. `Multiset` has the important property that inserting a new element into a `multiset` does not invalidate iterators that point to existing elements. Erasing an element from a multiset also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.
8	`map`	`Map` is a sorted associative container that associates objects of type `Key` with objects of type `Data`. `Map` is a **pair associative container**, meaning that its value type is `pair<const Key, Data>`. It is also a **unique associative container**, meaning that no two elements have the same key. `Map` has the important property that inserting a new element into a `map` does not invalidate iterators that point to existing elements. Erasing an element from a `map` also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.
9	`multimap`	`Multimap` is a **sorted associative container** that associates objects of type `Key` with objects of type `Data`. `multimap` is a pair associative container, meaning that its value type is `pair<const Key, Data>`. It is also a **multiple associative container**, meaning that there is no limit on the number of elements with the same key. `Multimap` has the important property that inserting a new element into a `multimap` does not invalidate iterators that point to existing elements. Erasing an element from a `multimap` also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.
Implementation dependent, non ANSI C++ (ISO/IEC C++)		
10	`hash`	The function object `hash<Type>` is a Hash Function; it is used as the default hash function by all of the Hashed Associative Containers that are included in the STL. The `hash<Type>` template is only defined for template arguments of type `char*`, `const char*`, `crope`, `wrope`, and the built-in integral types. If you need a Hash Function with a different argument type, you must either provide your own template specialization or else use a different Hash Function. This is implementation extension, not the ANSI C++ standard.
11	`hash_set`	`Hash_set` is a **hashed associative container** that stores objects of type `Key`. `Hash_set` is a **simple associative container**, meaning that its value type, as

		well as its key type, is `Key`. It is also a **unique associative container**, meaning that no two elements compare equal using the Binary Predicate `EqualKey`. `Hash_set` is useful in applications where it is important to be able to search for an element quickly. If it is important for the elements to be in a particular order, however, then `set` is more appropriate.
12	`hash_multiset`	`hash_multiset` is a **hashed associative container** that stores objects of type `Key`. `hash_multiset` is a **simple associative container**, meaning that its value type, as well as its key type, is `Key`. It is also a **multiple associative container**, meaning that two or more elements may compare equal using the Binary Predicate `EqualKey`. `hash_multiset` is useful in applications where it is important to be able to search for an element quickly. If it is important for the elements to be in a particular order, however, then `multiset` is more appropriate.
13	`hash_map`	`Hash_map` is a **hashed associative container** that associates objects of type `Key` with objects of type `Data`. `Hash_map` is a **pair associative container**, meaning that its value type is `pair<const Key, Data>`. It is also a **unique associative container**, meaning that no two elements have keys that compare equal using `EqualKey`. Looking up an element in a `hash_map` by its key is efficient, so `hash_map` is useful for "dictionaries" where the order of elements is irrelevant. If it is important for the elements to be in a particular order, however, then `map` is more appropriate.
14	`hash_multimap`	`Hash_multimap` is a **hashed associative container** that associates objects of type `Key` with objects of type `Data`. `Hash_multimap` is a **pair associative container**, meaning that its value type is `pair<const Key, Data>`. It is also a **multiple associative container**, meaning that there is no limit on the number of elements whose keys may compare equal using `EqualKey`. Looking up an element in a `hash_multimap` by its key is efficient, so `hash_multimap` is useful for "dictionaries" where the order of elements is irrelevant. If it is important for the elements to be in a particular order, however, then `multimap` is more appropriate.

Table 29.31

- Program example compiled using **`g++`**.

```
//******mapconstructor.cpp********
//map, constructor
//compiled with VC++ 7.0
//or .Net
#include <map>
#include <iostream>
using namespace std;

int main( )
{
   typedef pair<int, int> Int_Pair;
   map<int, int>::iterator mp0_Iter, mp1_Iter, mp3_Iter, mp4_Iter, mp5_Iter, mp6_Iter;
   map<int, int, greater<int> >::iterator mp2_Iter;

   //Create an empty map mp0 of key type integer
   map <int, int> mp0;

   //Create an empty map mp1 with the key comparison
   //function of less than, then insert 6 elements
   map <int, int, less<int> > mp1;
   mp1.insert(Int_Pair(1, 13));
   mp1.insert(Int_Pair(3, 23));
   mp1.insert(Int_Pair(3, 31));
   mp1.insert(Int_Pair(2, 23));
   mp1.insert(Int_Pair(6, 15));
   mp1.insert(Int_Pair(9, 25));

   //Create an empty map mp2 with the key comparison
   //function of greater than, then insert 3 elements
   map <int, int, greater<int> > mp2;
   mp2.insert(Int_Pair(3, 12));
   mp2.insert(Int_Pair(1, 31));
   mp2.insert(Int_Pair(2, 21));

   //Create a map mp3 with the
```

```
   //allocator of map mp1
   map <int, int>::allocator_type mp1_Alloc;
   mp1_Alloc = mp1.get_allocator();
   map <int, int> mp3(less<int>(), mp1_Alloc);
   mp3.insert(Int_Pair(1, 10));
   mp3.insert(Int_Pair(2, 12));

   //Create a copy, map mp4, of map mp1
   map <int, int> mp4(mp1);

   //Create a map mp5 by copying the range mp1[_First, _Last)
   map <int, int>::const_iterator mp1_PIter, mp1_QIter;
   mp1_PIter = mp1.begin();
   mp1_QIter = mp1.begin();
   mp1_QIter++;
   mp1_QIter++;
   map <int, int> mp5(mp1_PIter, mp1_QIter);

   //Create a map mp6 by copying the range mp4[_First, _Last)
   //and with the allocator of map mp2
   map <int, int>::allocator_type mp2_Alloc;
   mp2_Alloc = mp2.get_allocator();
   map <int, int> mp6(mp4.begin(), ++mp4.begin(), less<int>(), mp2_Alloc);

   //-------------------------------------------------------
   cout<<"Operation: map <int, int> mp0\n";
   cout<<"mp0 data: ";
   for(mp0_Iter = mp0.begin(); mp0_Iter != mp0.end(); mp0_Iter++)
      cout<<" "<<mp0_Iter->second;
   cout<<endl;

   cout<<"\nOperation1: map <int, int, less<int> > mp1\n";
   cout<<"Operation2: mp1.insert(Int_Pair(1, 13))...\n";
   cout<<"mp1 data: ";
   for(mp1_Iter = mp1.begin(); mp1_Iter != mp1.end(); mp1_Iter++)
      cout<<" "<<mp1_Iter->second;
   cout<<endl;

   cout<<"\nOperation1: map <int, int, greater<int> > mp2\n";
   cout<<"Operation2: mp2.insert(Int_Pair(3, 12))...\n";
   cout<<"mp2 data: ";
   for(mp2_Iter = mp2.begin(); mp2_Iter != mp2.end(); mp2_Iter++)
      cout<<" "<<mp2_Iter->second;
   cout<<endl;

   cout<<"\nOperation1:  map <int, int> mp3(less<int>(), mp1_Alloc)\n";
   cout<<"Operation2: mp3.insert(Int_Pair(1, 10))...\n";
   cout<<"mp3 data: ";
   for(mp3_Iter = mp3.begin(); mp3_Iter != mp3.end(); mp3_Iter++)
      cout<<" "<<mp3_Iter->second;
   cout<<endl;

   cout<<"\nOperation: map <int, int> mp4(mp1)\n";
     cout<<"mp4 data: ";
   for(mp4_Iter = mp4.begin(); mp4_Iter != mp4.end(); mp4_Iter++)
      cout<<" "<<mp4_Iter->second;
   cout<<endl;

   cout<<"\nOperation: map <int, int> mp5(mp1_PIter, mp1_QIter)\n";
   cout<<"mp5 data: ";
   for(mp5_Iter = mp5.begin(); mp5_Iter != mp5.end(); mp5_Iter++)
      cout<<" "<<mp5_Iter->second;
   cout<<endl;

   cout<<"\nOperation: map <int, int> mp6(mp4.begin(), \n++mp4.begin(), less<int>(),
mp2_Alloc);\n";
   cout<<"mp6 data: ";
   for(mp6_Iter = mp6.begin(); mp6_Iter != mp6.end(); mp6_Iter++)
      cout<<" "<<mp6_Iter->second;
   cout<<endl;
   return 0;
}
```

[bodo@bakawali ~]$ g++ mapconstructor.cpp -o mapconstructor
[bodo@bakawali ~]$ ./mapconstructor

```
Operation: map <int, int> mp0
mp0 data:
```

```
Operation1: map <int, int, less<int> > mp1
Operation2: mp1.insert(Int_Pair(1, 13))...
mp1 data:  13 23 23 15 25

Operation1: map <int, int, greater<int> > mp2
Operation2: mp2.insert(Int_Pair(3, 12))...
mp2 data:  12 21 31

Operation1:  map <int, int> mp3(less<int>(), mp1_Alloc)
Operation2: mp3.insert(Int_Pair(1, 10))...
mp3 data:  10 12

Operation: map <int, int> mp4(mp1)
mp4 data:  13 23 23 15 25

Operation: map <int, int> mp5(mp1_PIter, mp1_QIter)
mp5 data:  13 23

Operation: map <int, int> mp6(mp4.begin(),
++mp4.begin(), less<int>(), mp2_Alloc);
mp6 data:  13
```

----------End of container----------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.