

**MODULE 28**  
**--THE STL--**  
**CONTAINER PART II**  
**list, set, multiset**

My Training Period:      hours

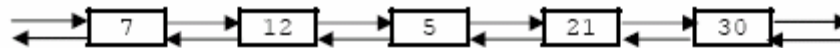
**Note:** Compiled using VC++7.0/.Net, win32 empty console mode application. **g++** program compilation examples given at the end of this Module.

**Abilities**

- Able to understand and use `list` sequence container.
- Able to understand and use `set` associative container.
- Able to understand and use `multiset` associative container.

**28.1 Lists**

- A list is implemented as a **doubly linked list** of elements. This means each element in a list has its own segment of memory and refers to its predecessor and its successor.
- Lists do not provide random access. It can be depicted as follow:



- For example, to access the tenth element, you must navigate the first nine elements by following the chain of their links. However, a step to the next or previous element is possible in constant time.
- Thus, the general access to an arbitrary element takes linear time (the average distance is proportional to the number of elements). This is a lot worse than the amortized constant time provided by vectors and deques.
- The advantage of a list is that the insertion or removal of an element is fast at any position. Only the links must be changed. This implies that moving an element in the middle of a list is very fast compared with moving an element in a vector or a deque.
- The list member functions `merge()`, `reverse()`, `unique()`, `remove()`, and `remove_if()` have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.
- List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid.
- The following general list example creates an empty list of characters, inserts all characters from 'a' to 'z', and prints all elements by using a loop that actually prints and removes the first element of the collection:

```
//list example
#include <iostream>
#include <list>
using namespace std;

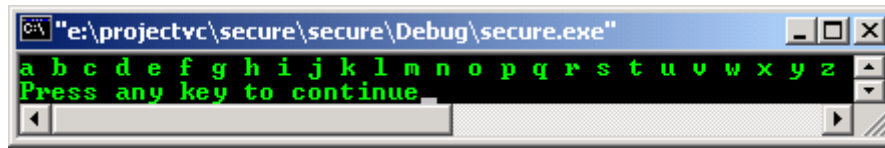
int main()
{
//list container for character elements
list<char> elem;

//append elements from 'a' to 'z'
for(char chs='a'; chs<= 'z'; ++chs)
elem.push_back(chs);

//while there are elements
//print and remove the element
while(!elem.empty())
{
cout<<elem.front()<<' ';
elem.pop_front();
}
cout<<endl;
```

```
return 0;
}
```

Output :



- With STL, using loop to print the outputs and removes the element is not a proper way. Normally, we would iterate over all elements using iterator. Using loop in the program example just for discussion.
- However, direct element access by using `operator[ ]` is not provided for lists. This is because lists don't provide random access, and thus using `operator[ ]` would cause bad performance.
- There is another way to loop over the elements and print them by using iterators.

## 28.2 <list> Header Members

### Operators

Operator	Description
<code>operator!=</code>	Tests if the list object on the left side of the operator is not equal to the list object on the right side.
<code>operator&lt;</code>	Tests if the list object on the left side of the operator is less than the list object on the right side.
<code>operator&lt;=</code>	Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.
<code>operator==</code>	Tests if the list object on the left side of the operator is equal to the list object on the right side.
<code>operator&gt;</code>	Tests if the list object on the left side of the operator is greater than the list object on the right side.
<code>operator&gt;=</code>	Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

Table 28.1

### list Template Class

Class	Description
list Class	A template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence.

table 28.2

- The STL list class is a template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence.
- The sequence is stored as a bidirectional linked list of elements, each containing a member of some type `Type`.

### list Template Class Members

#### Typedefs

Typedef	Description
<code>allocator_type</code>	A type that represents the allocator class for a list object.
<code>const_iterator</code>	A type that provides a bidirectional iterator that can read a <code>const</code> element in a list.
<code>const_pointer</code>	A type that provides a pointer to a <code>const</code> element in a list.
<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a list for reading and performing <code>const</code> operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any <code>const</code> element in a list.
<code>difference_type</code>	A type that provides the difference between two iterators those refer to

	elements within the same list.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a list.
pointer	A type that provides a pointer to an element in a list.
reference	A type that provides a reference to a <code>const</code> element stored in a list for reading and performing <code>const</code> operations.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed list.
size_type	A type that counts the number of elements in a list.
value_type	A type that represents the data type stored in a list.

Table 28.3

### list Template Class Member Functions

Member function	Description
<code>assign()</code>	Erases elements from a list and copies a new set of elements to the target list.
<code>back()</code>	Returns a reference to the last element of a list.
<code>begin()</code>	Returns an iterator addressing the first element in a list.
<code>clear()</code>	Erases all the elements of a list.
<code>empty()</code>	Tests if a list is empty.
<code>end()</code>	Returns an iterator that addresses the location succeeding the last element in a list.
<code>erase()</code>	Removes an element or a range of elements in a list from specified positions.
<code>front()</code>	Returns a reference to the first element in a list.
<code>get_allocator()</code>	Returns a copy of the allocator object used to construct a list.
<code>insert()</code>	Inserts an element or a number of elements or a range of elements into a list at a specified position.
<code>list()</code>	<code>list</code> 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 list.
<code>max_size()</code>	Returns the maximum length of a list.
<code>merge()</code>	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
<code>pop_back()</code>	Deletes the element at the end of a list.
<code>pop_front()</code>	Deletes the element at the beginning of a list.
<code>push_back()</code>	Adds an element to the end of a list.
<code>push_front()</code>	Adds an element to the beginning of a list.
<code>rbegin()</code>	Returns an iterator addressing the first element in a reversed list.
<code>remove()</code>	Erases elements in a list that match a specified value.
<code>remove_if()</code>	Erases elements from the list for which a specified predicate is satisfied.
<code>rend()</code>	Returns an iterator that addresses the location succeeding the last element in a reversed list.
<code>resize()</code>	Specifies a new size for a list.
<code>reverse()</code>	Reverses the order in which the elements occur in a list.
<code>size()</code>	Specifies a new size for a list.
<code>sort()</code>	Arranges the elements of a list in ascending order or with respect to some other order relation.
<code>splice()</code>	Removes elements from the argument list and inserts them into the target list.
<code>swap()</code>	Exchanges the elements of two lists.
<code>unique()</code>	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.

Table 28.4

- `list` 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 all or part of some other list.
- All constructors store an allocator object and initialize the list.
- `get_allocator()` returns a copy of the allocator object used to construct a list.
- None of the constructors perform any interim reallocations.

```
//list constructors
#include <list>
#include <iostream>
using namespace std;
```

```

int main()
{
list <int>::iterator li0Iter, li1Iter, li2Iter, li3Iter, li4Iter, li5Iter, li6Iter;

//Create an empty list li0
list <int> li0;

//Create a list li1 with 10 elements of default value 0
list <int> li1(10);

//Create a list li2 with 8 elements of value 7
list <int> li2(8, 7);

//Create a list li3 with 9 elements of value 8 and with the
//allocator of list li2
list <int> li3(9, 8, li2.get_allocator());

//li4, a copy of list li2
list <int> li4(li2);

//Create a list li5 by copying the range of li4[_First, _Last)
li4Iter = li4.begin();
li4Iter++;
li4Iter++;
li4Iter++;
li4Iter++;
list <int> li5(li4.begin(), li4Iter);

//Create a list li6 by copying the range of li4[_First, _Last) and with
//the allocator of list li2
li4Iter = li4.begin();
li4Iter++;
li4Iter++;
li4Iter++;
list <int> li6(li4.begin(), li4Iter, li2.get_allocator());

//-----
cout<<"Operation: list <int> li0\n";
cout<<"li0 data: ";
for(li0Iter = li0.begin(); li0Iter != li0.end(); li0Iter++)
cout<<" "<<*li0Iter;
cout<<endl;

cout<<"\nOperation: list <int> li1(10)\n";
cout<<"li1 data: ";
for(li1Iter = li1.begin(); li1Iter != li1.end(); li1Iter++)
cout<<" "<<*li1Iter;
cout<<endl;

cout<<"\nOperation: list <int> li2(8, 7)\n";
cout<<"li2 data: ";
for(li2Iter = li2.begin(); li2Iter != li2.end(); li2Iter++)
cout<<" "<<*li2Iter;
cout<<endl;

cout<<"\nOperation: list <int> li3(9, 8, li2.get_allocator())\n";
cout<<"li3 data: ";
for(li3Iter = li3.begin(); li3Iter != li3.end(); li3Iter++)
cout<<" "<<*li3Iter;
cout<<endl;

cout<<"\nOperation: list <int> li4(li2);\n";
cout<<"li4 data: ";
for(li4Iter = li4.begin(); li4Iter != li4.end(); li4Iter++)
cout<<" "<<*li4Iter;
cout<<endl;

cout<<"\nOperation1: li4Iter = li4.begin(), li4Iter++...\n";
cout<<"Operation2: list <int> li5(li4.begin(), li4Iter)\n";
cout<<"li5 data: ";
for(li5Iter = li5.begin(); li5Iter != li5.end(); li5Iter++)
cout<<" "<<*li5Iter;
cout<<endl;

cout<<"\nOperation1: li4Iter = li4.begin(), li4Iter++...\n";
cout<<"Operation2: list <int> li6(li4.begin(), li4Iter,\n"
"    li2.get_allocator())\n";
cout<<"li6 data: ";
for(li6Iter = li6.begin(); li6Iter != li6.end(); li6Iter++)
cout<<" "<<*li6Iter;

```

```

cout<<endl;
return 0;
}

```

Output:

- The return value is the first insert ( ) function returns an iterator that point to the position where the new element was inserted into the list.
- Any insertion operation can be expensive in term of time and resource.

```

//list, insert()
#include <list>
#include <iostream>
using namespace std;

int main()
{
list <int> lis1, lis2;
list <int>::iterator Iter;

lis1.push_back(13);
lis1.push_back(22);
lis1.push_back(15);
lis2.push_back(9);
lis2.push_back(5);
lis2.push_back(45);

cout<<"lis1 data: ";
for(Iter = lis1.begin(); Iter != lis1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"lis2 data: ";
for(Iter = lis2.begin(); Iter != lis2.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation1: lis1.begin() then Iter++...\n";
cout<<"Operation2: lis1.insert(Iter, 55)\n";
Iter = lis1.begin();
Iter++;
lis1.insert(Iter, 55);
cout<<"lis1 data: ";
for(Iter = lis1.begin(); Iter != lis1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation1: lis1.begin() then Iter++...\n";
cout<<"Operation2: lis1.insert(Iter, 3, 30)\n";

```

```

Iter = lis1.begin();
Iter++;
Iter++;
lis1.insert(Iter, 3, 30);

cout<<"lis1 data: ";
for(Iter = lis1.begin(); Iter != lis1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation2: lis1.insert(++lis1.begin(),\n"
"    lis2.begin(),--lis2.end())\n";
lis1.insert(++lis1.begin(), lis2.begin(),--lis2.end());
cout<<"lis1 data: ";
for(Iter = lis1.begin(); Iter != lis1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;
return 0;
}

```

**Output:**

```

lis1 data: 13 22 15
lis2 data: 9 5 45

Operation1: lis1.begin() then Iter++...
Operation2: lis1.insert(Iter, 55)
lis1 data: 13 55 22 15

Operation1: lis1.begin() then Iter++...
Operation2: lis1.insert(Iter, 3, 30)
lis1 data: 13 55 30 30 30 22 15

Operation2: lis1.insert(++lis1.begin(),
    lis2.begin(),--lis2.end())
lis1 data: 13 9 5 55 30 30 30 22 15
Press any key to continue

```

- The order of the elements remaining is not affected.

```

//list, remove()
#include <list>
#include <iostream>
using namespace std;

int main( )
{
list <int> lis1;
list <int>::iterator lis1Iter, lis2Iter;

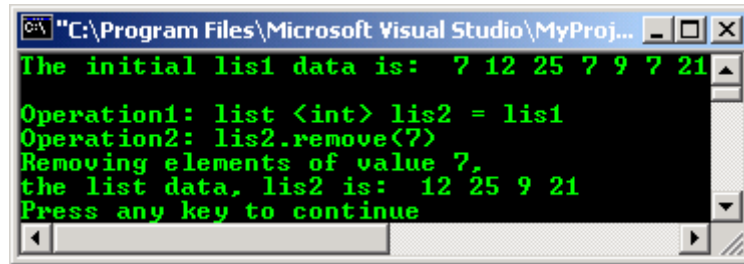
lis1.push_back(7);
lis1.push_back(12);
lis1.push_back(25);
lis1.push_back(7);
lis1.push_back(9);
lis1.push_back(7);
lis1.push_back(21);

cout<<"The initial lis1 data is: ";
for(lis1Iter = lis1.begin(); lis1Iter != lis1.end(); lis1Iter++)
cout<<" "<<*lis1Iter;
cout<<endl;

cout<<"\nOperation1: list <int> lis2 = lis1\n";
cout<<"Operation2: lis2.remove(7)\n";
list <int> lis2 = lis1;
lis2.remove(7);
cout<<"Removing elements of value 7, \nthe list data, lis2 is: ";
for(lis2Iter = lis2.begin(); lis2Iter != lis2.end(); lis2Iter++)
cout<<" "<<*lis2Iter;
cout<<endl;
return 0;
}

```

Output :



```
"C:\Program Files\Microsoft Visual Studio\MyProj...
The initial lis1 data is: 7 12 25 7 9 7 21
Operation1: list <int> lis2 = lis1
Operation2: lis2.remove(7)
Removing elements of value 7,
the list data, lis2 is: 12 25 9 21
Press any key to continue
```

- The first member function puts the elements in ascending order by default.
- The member template function orders the elements according to the user-specified comparison operation `_Comp` of class `Traits`.

```
//list, sort()
#include <list>
#include <iostream>
using namespace std;

int main()
{
    list <int> ls1;
    list <int>::iterator ls1Iter;

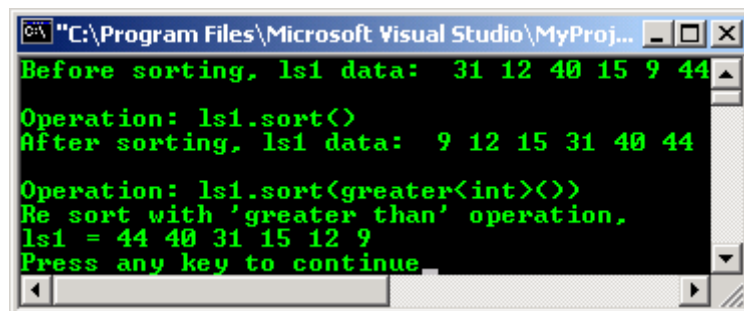
    ls1.push_back(31);
    ls1.push_back(12);
    ls1.push_back(40);
    ls1.push_back(15);
    ls1.push_back(9);
    ls1.push_back(44);

    cout<<"Before sorting, ls1 data: ";
    for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
        cout<<" "<<*ls1Iter;
    cout<<endl;

    cout<<"\nOperation: ls1.sort()\n";
    ls1.sort();
    cout<<"After sorting, ls1 data: ";
    for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
        cout<<" "<<*ls1Iter;
    cout<<endl;

    cout<<"\nOperation: ls1.sort(greater<int>())\n";
    ls1.sort(greater<int>());
    cout<<"Re sort with 'greater than' operation,\nls1 = ";
    for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
        cout<<" "<<*ls1Iter;
    cout<<endl;
    return 0;
}
```

Output :



```
"C:\Program Files\Microsoft Visual Studio\MyProj...
Before sorting, ls1 data: 31 12 40 15 9 44
Operation: ls1.sort()
After sorting, ls1 data: 9 12 15 31 40 44
Operation: ls1.sort(greater<int>())
Re sort with 'greater than' operation,
ls1 = 44 40 31 15 12 9
Press any key to continue
```

```
//list, splice()
#include <list>
#include <iostream>
using namespace std;
```

```

int main( )
{
list <int> ls1, ls2, ls3, ls4;
list <int>::iterator ls1Iter, ls2Iter, ls3Iter, ls4Iter, PIter, QIter, RIter;

ls1.push_back(7);
ls1.push_back(15);
ls2.push_back(9);
ls2.push_back(22);
ls2.push_back(12);
ls3.push_back(29);
ls3.push_back(30);
ls4.push_back(33);
ls4.push_back(25);
ls4.push_back(51);

cout<<"ls1 data: ";
for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
cout<<" "<<*ls1Iter;
cout<<endl;

cout<<"ls2 data: ";
for(ls2Iter = ls2.begin(); ls2Iter != ls2.end(); ls2Iter++)
cout<<" "<<*ls2Iter;
cout<<endl;

cout<<"ls3 data: ";
for(ls3Iter = ls3.begin(); ls3Iter != ls3.end(); ls3Iter++)
cout<<" "<<*ls3Iter;
cout<<endl;

cout<<"ls4 data: ";
for(ls4Iter = ls4.begin(); ls4Iter != ls4.end(); ls4Iter++)
cout<<" "<<*ls4Iter;
cout<<endl;

cout<<"\nOperation: ls2.splice(PIter, ls1)\n";
PIter = ls2.begin();
PIter++;
ls2.splice(PIter, ls1);
cout<<"ls2 data, after splicing \nls1 into ls2: ";
for(ls2Iter = ls2.begin(); ls2Iter != ls2.end(); ls2Iter++)
cout<<" "<<*ls2Iter;
cout<<endl;

cout<<"\nOperation: ls2.splice(PIter, ls3, QIter)\n";
QIter = ls3.begin();
ls2.splice(PIter, ls3, QIter);
cout<<"ls2 data, after splicing the first \nelement of ls3 into ls2: ";
for(ls2Iter = ls2.begin(); ls2Iter != ls2.end(); ls2Iter++)
cout<<" "<<*ls2Iter;
cout<<endl;

cout<<"\nOperation: ls2.splice(PIter, ls4, QIter, RIter)\n";
QIter = ls4.begin();
RIter = ls4.end();
RIter--;
ls2.splice(PIter, ls4, QIter, RIter);
cout<<"ls2 data, after splicing a range \nof ls4 into ls2: ";
for(ls2Iter = ls2.begin(); ls2Iter != ls2.end(); ls2Iter++)
cout<<" "<<*ls2Iter;
cout<<endl;
return 0;
}

```

**Output:**



```

"C:\Program Files\Microsoft Visual Studio\MyProjects\...
ls1 data: 7 15
ls2 data: 9 22 12
ls3 data: 29 30
ls4 data: 33 25 51

Operation: ls2.splice(PIter, ls1)
ls2 data, after splicing
ls1 into ls2: 9 7 15 22 12

Operation: ls2.splice(PIter, ls3, QIter)
ls2 data, after splicing the first
element of ls3 into ls2: 9 7 15 29 22 12

Operation: ls2.splice( PIter, ls4, QIter, RIter)
ls2 data, ffter splicing a range
of ls4 into ls2: 9 7 15 29 33 25 22 12
Press any key to continue

```

- This function assumes that the list is sorted, so that all duplicate elements are adjacent. Duplicates that are not adjacent will not be deleted.
- The first member function removes every element that compares equal to its preceding element.
- The second member function removes every element that satisfies the predicate function when compared with its preceding element.

```

//list, unique()
#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<int> ls1;
    list<int>::iterator ls1Iter, ls2Iter, ls3Iter;
    not_equal_to<int> mypred;

    ls1.push_back(-12);
    ls1.push_back(12);
    ls1.push_back(12);
    ls1.push_back(22);
    ls1.push_back(22);
    ls1.push_back(13);
    ls1.push_back(-12);
    ls1.push_back(14);

    cout<<"ls1 data, the initial list:\n";
    for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
        cout<<" "<<*ls1Iter;
    cout<<endl;

    cout<<"\nOperation1: list<int> ls2 = ls1\n";
    cout<<"Operation2: ls2.unique()\n";
    list<int> ls2 = ls1;
    ls2.unique();
    cout<<"ls2 data, after removing successive duplicate\nelements: ";
    for(ls2Iter = ls2.begin(); ls2Iter != ls2.end(); ls2Iter++)
        cout<<" "<<*ls2Iter;
    cout<<endl;

    cout<<"\nOperation1: list<int> ls3 = ls2\n";
    cout<<"Operation2: ls3.unique(mypred)\n";
    list<int> ls3 = ls2;
    ls3.unique(mypred);
    cout<<"ls3 data, after removing successive unequal\nelements: ";
    for(ls3Iter = ls3.begin(); ls3Iter != ls3.end(); ls3Iter++)
        cout<<" "<<*ls3Iter;
    cout<<endl;
    return 0;
}

```

**Output:**

```

"C:\Program Files\Microsoft Visual Studio\MyProject...
ls1 data, the initial list:
-12 12 12 22 22 13 -12 14

Operation1: list <int> ls2 = ls1
Operation2: ls2.unique()
ls2 data, after removing successive duplicate
elements: -12 12 22 13 -12 14

Operation1: list <int> ls3 = ls2
Operation2: ls3.unique(mypred)
ls3 data, after removing successive unequal
elements: -12 -12
Press any key to continue

```

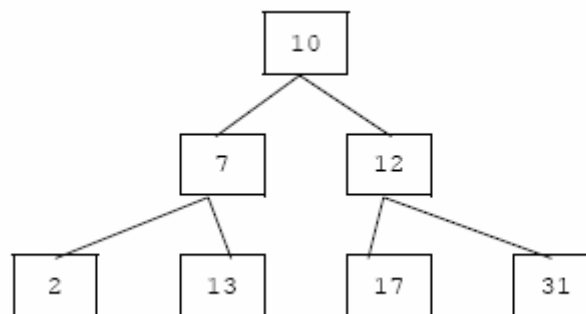
-----End of list-----  
---www.tenouk.com---

### Further reading and digging:

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

### 28.3 Associative Containers

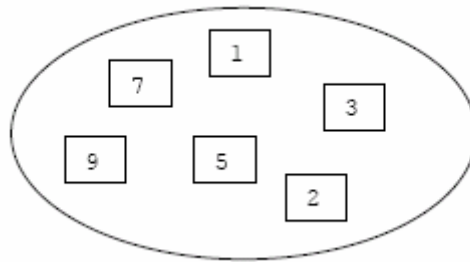
- Associative containers sort their elements automatically according to a certain ordering criterion.
- This criterion takes the form of a function that **compares** either the **value** or a **special key** that is defined for the value.
- By default, the containers compare the elements or the keys with operator less than (<). However, you can supply your own comparison function to define another ordering criterion.
- Associative containers are typically implemented as **binary trees**. Thus, every element (every node) has one parent and two children. All ancestors to the **left** have **lesser values**; all ancestors to the **right** have **greater values**. It can be depicted as follow:



- The associative containers differ in the **kind of elements** they support and **how they handle duplicates**. The following is discussion of the associative containers that are predefined in the STL.
- All of the associative container classes have an optional template argument for the sorting criterion. **The default sorting criterion is the operator < (les than)**.
- The sorting criterion is also used as the test for equality; that is, two elements are equal if neither is less than the other.
- You can consider a set as a special kind of map, in which the **value** is identical to the **key**. In fact, all of these associative container types are usually implemented by using the same basic implementation of a binary tree.
- The choice of container type should be based in general on the type of searching and inserting required by the application.
- Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container.
- Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

## 28.4 Sets

- A set is a collection in which elements are **sorted according to their own values**. Each element may occur only once, thus duplicates are not allowed. Its structure can be depicted as shown below.



## 28.5 <set> Header Members

### Operators

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

Table 28.5

### set Specialized Template Functions

Specialized template function	Description
swap()	Exchanges the elements of two sets or multisets.

Table 28.6

### set Class Templates

Class	Description
set Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered.
multiset Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered.

Table 28.7

### set Class Template Members

#### Typedefs

Typedef	Description
allocator_type	A type that represents the allocator class for the set object.
const_iterator	A type that provides a bidirectional iterator that can read a const element in the set.
const_pointer	A type that provides a pointer to a const element in a set.

<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a set for reading and performing <code>const</code> operations.
<code>const_reverse_iterator</code>	A type that provides a bidirectional iterator that can read any <code>const</code> element in the set.
<code>difference_type</code>	A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.
<code>iterator</code>	A type that provides a bidirectional iterator that can read or modify any element in a set.
<code>key_compare</code>	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.
<code>key_type</code>	The type describes an object stored as an element of a set in its capacity as sort key.
<code>pointer</code>	A type that provides a pointer to an element in a set.
<code>reference</code>	A type that provides a reference to an element stored in a set.
<code>reverse_iterator</code>	A type that provides a bidirectional iterator that can read or modify an element in a reversed set.
<code>size_type</code>	An unsigned integer type that can represent the number of elements in a set.
<code>value_compare</code>	The type that provides a function object that can compare two elements to determine their relative order in the set.
<code>value_type</code>	The type describes an object stored as an element of a set in its capacity as a value.

Table 28.8

### set Class Template Member Functions

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

Table 28.9

- The STL container class `set` is used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered.

- The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

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

## Parameters

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

Table 28.10

- An STL set is:
  - An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value. 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.
  - 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. Since set is also a simple associative container, its elements are also unique.
- A set is also described as a template class because the functionality it provides is generic and independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.
- The elements of a set are unique and serve as their own sort keys. 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 multiset would be the appropriate container structure.
- If unique definitions were attached as values to the list of key words, then a map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a multimap would be the container of choice.
- The set 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`.
- In general, the elements need to be merely less than comparable to establish this order so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements.
- The iterator provided by the set class is a bidirectional iterator, but the class member functions `insert()` and `set()` 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.
- It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence.
- This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`_First`, `_Last`) in the context of the class's member functions.

## set Constructor

- Constructs a set that is empty or that is a copy of all or part of some other set. The following is a code for set constructor `set::set`. It is provided here for parameters terminologies reference and **will not be repeated** for other associative containers.

```

set( );
explicit set(
    const Traits& _Comp
);
explicit set(
    const Traits& _Comp,
    const Allocator& _Al
);
set(
    const _set& _Right
);
template<class InputIterator>
set(
    InputIterator _First,
    InputIterator _Last
);
template<class InputIterator>
set(
    InputIterator _First,
    InputIterator _Last,
    const Traits& _Comp
);
template<class InputIterator>
set(
    InputIterator _First,
    InputIterator _Last,
    const Traits& _Comp,
    const Allocator& _Al
);

```

## Parameters

Parameter	Description
<code>_Al</code>	The storage allocator class to be used for this set object, which defaults to <code>Allocator</code> .
<code>_Comp</code>	The comparison function of type <code>const Traits</code> used to order the elements in the set, which defaults to <code>Compare</code> .
<code>_Right</code>	The set of which the constructed set is to be a copy.
<code>_First</code>	The position of the first element in the range of elements to be copied.
<code>_Last</code>	The position of the first element beyond the range of elements to be copied.

Table 28.11

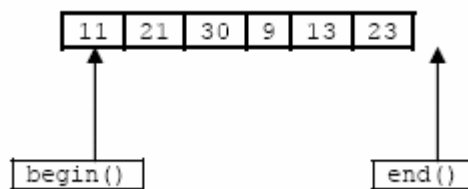
- All constructors store a type of allocator object that manages memory storage for the set and that can later be returned by calling `get_allocator()`. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.
- All constructors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

## Note:

- You will find somewhere sometime in STL the half-open ranges format as shown below.

`[begin, end)`            or            `[begin, end[`

- The range is defined so that it includes the position used as the beginning of the range **but** excludes the position used as the end. It can be illustrated below:



- From the illustration, `begin()` and `end()` define a half-open range that includes the first element but excludes the last. A half-open range has two advantages:
  1. You have a simple end criterion for loops that iterate over the elements, they simply continue as long as `end()` is not reached.
  2. It avoids special handling for empty ranges. For empty ranges, `begin()` is equal to `end()`.

```
//set, constructor
#include <set>
#include <iostream>
using namespace std;

char main()
{
    set <char>::iterator st0_Iter, st1_Iter, st2_Iter, st3_Iter, st4_Iter, st5_Iter, st6_Iter;

    //Create an empty set st0 of key type char
    set <char> st0;

    //Create an empty set st1 with the key comparison
    //function of less than, then insert 6 elements
    set <char, less<char> > st1;
    st1.insert('p');
    st1.insert('e');
    st1.insert('g');
    st1.insert('P');
    st1.insert('y');
    st1.insert('b');

    //Create an empty set st2 with the key comparison
    //function of greater than, then insert 2 elements
    set <char, greater<char> > st2;
    st2.insert('f');
    st2.insert('k');

    //Create a set st3 with the
    //allocator of set st1
    set <char>::allocator_type st1_Alloc;
    st1_Alloc = st1.get_allocator();
    set <char> st3(less<char>(), st1_Alloc);
    st3.insert('u');
    st3.insert('U');

    //set st4, a copy of set st1
    set <char> st4(st1);

    //Create a set st5 by copying the range st1[_First, _Last)
    set <char>::const_iterator st1_PIter, st1_QIter;
    st1_PIter = st1.begin();
    st1_QIter = st1.begin();
    st1_QIter++;
    st1_QIter++;
    st1_QIter++;
    set <char> st5(st1_PIter, st1_QIter);

    //Create a set st6 by copying the range st4[_First, _Last)
    //and with the allocator of set st2
    set <char>::allocator_type st2_Alloc;
    st2_Alloc = st2.get_allocator();
    set <char> st6(st4.begin(), ++st4.begin(), less<char>(), st2_Alloc);

    //-----
    cout<<"Operation: set <char> st0\n";
    cout<<"st0 data: ";
    for(st0_Iter = st0.begin(); st0_Iter != st0.end(); st0_Iter++)
        cout<<" "<<*st0_Iter;
    cout<<endl;
}
```

```

cout<<"\nOperation1: set <char, less<char> > st1\n";
cout<<"Operation2: st1.insert('p')...\n";
cout<<"st1 data: ";
for(st1_Iter = st1.begin(); st1_Iter != st1.end(); st1_Iter++)
    cout<<" "<<*st1_Iter;
cout<<endl;

cout<<"\nOperation1: set <char, greater<char> > st2\n";
cout<<"Operation2: st2.insert('f')...\n";
cout<<"st2 data: "<<*st2.begin()<<" "<<*++st2.begin()<<endl;

cout<<"\nOperation1: set <char> st3(less<char>(), st1_Alloc)\n";
cout<<"Operation2: st3.insert('u')\n";
cout<<"st3 data: ";
for(st3_Iter = st3.begin(); st3_Iter != st3.end(); st3_Iter++)
    cout<<" "<<*st3_Iter;
cout<<endl;

cout<<"\nOperation: set <char> st4(st1)\n";
cout<<"st4 data: ";
for(st4_Iter = st4.begin(); st4_Iter != st4.end(); st4_Iter++)
    cout<<" "<<*st4_Iter;
cout<<endl;

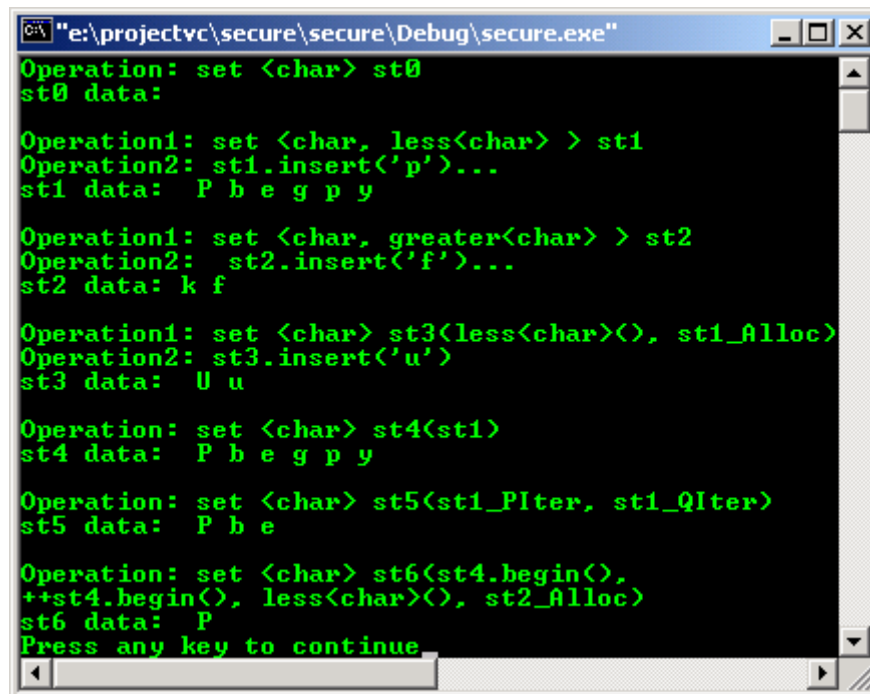
cout<<"\nOperation: set <char> st5(st1_PIter, st1_QIter)\n";
cout<<"st5 data: ";
for(st5_Iter = st5.begin(); st5_Iter != st5.end(); st5_Iter++)
    cout<<" "<<*st5_Iter;
cout<<endl;

cout<<"\nOperation: set <char> st6(st4.begin(), \n++st4.begin(), less<char>(),
                                st2_Alloc)\n";

cout<<"st6 data: ";
for(st6_Iter = st6.begin(); st6_Iter != st6.end(); st6_Iter++)
    cout<<" "<<*st6_Iter;
cout<<endl;
return 0;
}

```

**Output:**



- The return value is 1 if the set contains an element whose sort key matches the parameter key. 0 if the set does not contain an element with a matching key.
- The member function returns the number of elements in the following range:

```
[lower_bound (_Key), upper_bound (_Key)).
```

```
//set, count()
```



```

//some warning during the compilation
#include <set>
#include <iostream>
using namespace std;

int main()
{
set <int> st1;
int i;

st1.insert(1);
st1.insert(2);
st1.insert(1);

//Keys must be unique in set, duplicates are ignored
i = st1.count(1);
cout<<"The number of elements in st1 with a sort key of 1 is: "<<i<<endl;

i = st1.count(2);
cout<<"The number of elements in st1 with a sort key of 2 is: "<<i<<endl;

i = st1.count(3);
cout<<"The number of elements in st1 with a sort key of 3 is: "<<i<<endl;

return 0;
}

```

### Output:

- The return value is a pair of iterators where the first is the lower\_bound of the key and the second is the upper\_bound of the key.
- To access the first iterator of a pair *pr* returned by the member function, use *pr.first*, and to dereference the lower bound iterator, use *\*(pr.first)*.
- To access the second iterator of a pair *pr* returned by the member function, use *pr.second*, and to dereference the upper bound iterator, use *\*(pr.second)*.

```

//set, equal_range()
//some warning during compilation
#include <set>
#include <iostream>
using namespace std;

int main()
{
typedef set<int, less<int> > IntSet;
IntSet st1;
set <int>::iterator stlIter;
set <int, less< int > > :: const_iterator stl_RcIter;

st1.insert(10);
st1.insert(20);
st1.insert(30);
st1.insert(40);
st1.insert(50);

cout<<"st1 data: ";
for(stlIter = st1.begin(); stlIter != st1.end(); stlIter++)
cout<<" "<<*stlIter;
cout<<endl;

pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
p1 = st1.equal_range(30);

cout<<"\nThe upper bound of the element with\n"
<<"a key of 30 in the set st1 is: "
<<*(p1.second)<<endl;

cout<<"\nThe lower bound of the element with\n"

```

```

    <<"a key of 30 in the set st1 is: "
    <<*(p1.first)<<endl;

//Compare the upper_bound called directly
st1_RcIter = st1.upper_bound(30);
cout<<"\nA direct call of upper_bound(30) gives "
    <<st1_RcIter<<" matching\nthe 2nd element of the pair"
    <<" returned by equal_range(30)."<<endl;

cout<<"\nOperation: p2 = st1.equal_range(60)\n";
p2 = st1.equal_range(60);

//If no match is found for the key,
//both elements of the pair return end()
if ((p2.first == st1.end()) && (p2.second == st1.end()))
cout<<"\nThe set st1 doesn't have an element\n"
    <<"with a key less than 60."<<endl;
else
cout<<"The element of set st1 with a key >= 60 is: "
    <<*(p1.first)<<endl;
return 0;
}

```

#### Output:

```

"C:\Program Files\Microsoft Visual Studio\MyProjects\seek\Debug\...
st1 data: 10 20 30 40 50

The upper bound of the element with
a key of 30 in the set st1 is: 40

The lower bound of the element with
a key of 30 in the set st1 is: 30

A direct call of upper_bound(30) gives 40 matching
the 2nd element of the pair returned by equal_range(30).

Operation: p2 = st1.equal_range(60)

The set st1 doesn't have an element
with a key less than 60.
Press any key to continue

```

- The return value is the function object that a set uses to order its elements, which is the template parameter Traits.
- The stored object defines the member function:

```

bool operator()(const Key& _xVal, const Key& _yVal);

```
- Which returns true if \_xVal precedes and is not equal to \_yVal in the sort order.
- Note that both key\_compare and value\_compare are synonyms for the template parameter Traits.
- Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

```

//set, key_comp()
#include <set>
#include <iostream>
using namespace std;

int main()
{
    set <int, less<int> > st1;
    set<int, less<int> >::key_compare kcl = st1.key_comp();
    bool res1 = kcl(3, 7);
    if(res1 == true)
    {
        cout<<"kcl(3,7) returns value of true, "
            <<"where kcl\nis the function object of st1."
            <<endl;
    }
    else
    {

```

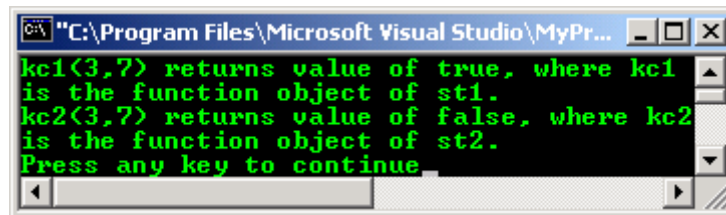
```

cout<<"kc1(3,7) returns value of false "
    <<"where kc1\nis the function object of st1."
    <<endl;
}

set <int, greater<int> > st2;
set<int, greater<int> >::key_compare kc2 = st2.key_comp();
bool res2 = kc2(3, 7);
if(res2 == true)
{
cout<<"kc2(3,7) returns value of true, "
    <<"where kc2\nis the function object of st2."
    <<endl;
}
else
{
cout<<"kc2(3,7) returns value of false, "
    <<"where kc2\nis the function object of st2."
    <<endl;
}
return 0;
}

```

**Output :**



```

"C:\Program Files\Microsoft Visual Studio\MyPr...
kc1(3,7) returns value of true, where kc1
is the function object of st1.
kc2(3,7) returns value of false, where kc2
is the function object of st2.
Press any key to continue

```

- The return value is an iterator or `const_iterator` that addresses the location of an element in a set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the set if no match is found for the key.

```

//set, lower_bound()
#include <set>
#include <iostream>
using namespace std;

int main( )
{
set <int> st1;
set <int> :: const_iterator stlIter, stl_PIter, stl_QIter;

st1.insert(11);
st1.insert(21);
st1.insert(30);
st1.insert(10);
st1.insert(22);

cout<<"st1 data: ";
for(stlIter = st1.begin(); stlIter != st1.end(); stlIter++)
cout<<" "<<*stlIter;
cout<<endl;

stl_QIter = st1.lower_bound(21);
cout<<"The element of set st1 with a key of 21 is: "
    <<*stl_QIter<<endl;

stl_QIter = st1.lower_bound(60);

//If no match is found for the key, end() is returned
if(stl_QIter == st1.end())
cout<<"The set st1 doesn't have an element "
    <<"with a key of 60."<<endl;
else
cout<<"The element of set st1 with a key of 60 is: "
    <<*stl_QIter<<endl;

//The element at a specific location in the set can be found
//by using a dereferenced iterator that addresses the location
stl_PIter = st1.end();
stl_PIter--;

```

```

stl_QIter = stl.lower_bound(*stl_PIter);
cout<<"The element of stl with a key matching "
    <<"that\nof the last element is: "
    <<*stl_QIter<<endl;
return 0;
}

```

**Output:**

```

"C:\Program Files\Microsoft Visual Studio\MyProjects\seek\De...
stl data: 10 11 21 22 30
The element of set stl with a key of 21 is: 21
The set stl doesn't have an element with a key of 60.
The element of stl with a key matching that
of the last element is: 30
Press any key to continue

```

- Return value is an iterator or const\_iterator that addresses the location of an element in a set that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the set if no match is found for the key.

```

//set, upper_bound()
#include <set>
#include <iostream>
using namespace std;

int main()
{
    set <int> st1;
    set <int> :: const_iterator stlIter, stlPIter, stlQIter;

    st1.insert(9);
    st1.insert(12);
    st1.insert(20);
    st1.insert(13);
    st1.insert(11);

    cout<<"stl data: ";
    for(stlIter = st1.begin(); stlIter != st1.end(); stlIter++)
        cout<<" "<<*stlIter;
    cout<<endl;

    stlQIter = st1.upper_bound(9);
    cout<<"The first element of set st1 with a key greater "
        <<"than 9 is: "<<*stlQIter<<endl;

    stlQIter = st1.upper_bound(22);

    //If no match is found for the key, end( ) is returned
    if(stlQIter == st1.end())
        cout<<"The set st1 doesn't have an element "
            <<"with a key greater than 22."<<endl;
    else
        cout<<"The element of set st1 with a key > 22 is: "
            <<*stlQIter<<endl;

    //The element at a specific location in the set can be found
    //by using a dereferenced iterator addressing the location
    stlPIter = st1.begin();
    stlQIter = st1.upper_bound(*stlPIter);
    cout<<"The first element of stl with a key greater than\n"
        <<"that of the initial element of stl is: "
        <<*stlQIter<<endl;
    return 0;
}

```

**Output:**

```

"C:\Program Files\Microsoft Visual Studio\MyProjects\seek\Debug\seek.exe"
st1 data: 9 11 12 13 20
The first element of set st1 with a key greater than 9 is: 11
The set st1 doesn't have an element with a key greater than 22.
The first element of st1 with a key greater than
that of the initial element of st1 is: 11
Press any key to continue

```

- The return value is a function object that a set uses to order its elements, which is the template parameter Traits.
- The stored object defines the member function:

```
bool operator(const Key& _xVal, const Key& _yVal);
```

- Which returns **true** if `_xVal` precedes and is not equal to `_yVal` in the sort order.
- Note that both `value_compare` and `key_compare` are synonyms for the template parameter Traits. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

```

//set, value_comp()
#include <set>
#include <iostream>
using namespace std;

int main()
{
    set <int, less<int> > st1;
    set <int, less<int> >::value_compare vcom1 = st1.value_comp();
    bool result1 = vcom1(5, 9);
    if(result1 == true)
    {
        cout<<"vcom1(5,9) returns value of true, "
            <<"\nwhere vcom1 is the function object of st1."
            <<endl;
    }
    else
    {
        cout<<"vcom1(5,9) returns value of false, "
            <<"\nwhere vcom1 is the function object of st1."
            <<endl;
    }

    set <int, greater<int> > st2;
    set<int, greater<int> >::value_compare vcom2 = st2.value_comp();
    bool result2 = vcom2(5, 9);
    if(result2 == true)
    {
        cout<<"vcom2(5,9) returns value of true, "
            <<"\nwhere vcom2 is the function object of st2."
            <<endl;
    }
    else
    {
        cout<<"vcom2(5,9) returns value of false, "
            <<"\nwhere vcom2 is the function object of st2."
            <<endl;
    }
    return 0;
}

```

Output :

```

"C:\Program Files\Microsoft Visual Studio\MyPr...
vcom1(5,9) returns value of true,
where vcom1 is the function object of st1.
vcom2(5,9) returns value of false,
where vcom2 is the function object of st2.
Press any key to continue

```

Note:

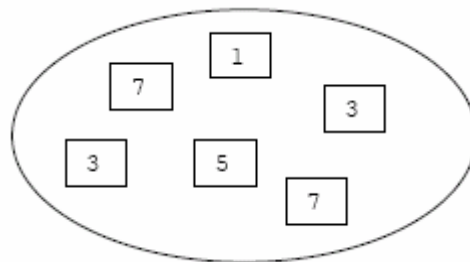
- For other associative containers, program examples as shown before will not be presented again, except the containers constructor, because they are similar.
- You can try on your own by using the same previous program examples, replace the containers and keep other codes as it is. Recompile and re run. Start using your own brain and creativity. Be creative :o).

-----End of set-----  
---www.tenouk.com---

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

## 28.6 Multiset

- A multiset is the same as a **set except that duplicates are allowed**. Thus, a multiset may contain multiple elements that have the same value. It can be depicted as follows:



- The elements of a multiset may be **multiple** and serve as their **own sort keys**, so **keys are not unique**. If the definitions were not unique, then a multimap would be the container of choice.
- The multiset orders the sequence it controls by calling a stored function object of type Compare. This stored object is a comparison function that may be accessed by calling the member function `key_comp()`.
- The iterator provided by the multiset class is a bidirectional iterator.

### multiset Members

### multiset Typedefs

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

	capacity as a value.
--	----------------------

Table 28.12

### multiset Member Functions

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

Table 28.13

### multiset Template Class

- The STL multiset class is used for the storage and retrieval of data from a collection in which **the values of the elements contained need not be unique** and in which they serve as the key values according to which the data is automatically ordered.
- The key value of an element in a multiset may not be changed directly. Instead, old values must be deleted and elements with new values inserted. The following code is the multiset template class.

```
template <
    class Key,
    class Compare = less<Key>,
    class Allocator = allocator<Key>
>
```

### Parameters

Parameter	Description
Key	The element data type to be stored in the multiset.
Compare	The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multiset. The binary predicate <code>less&lt;Key&gt;</code> is the default value.
Allocator	The type that represents the stored allocator object that encapsulates details about the multiset's allocation and de-allocation of memory. The default value is <code>allocator&lt;Key&gt;</code> .

Table 28.14

- The STL multiset class is:
  - An associative container, which is 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 in the sense that its elements do not need to have unique keys, so that one key value can have many element values associated with it.
  - A simple associative container because its element values are 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. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

### multiset Constructor

- Constructs a multiset that is empty or that is a copy of all or part of some other multiset.
- All constructors store a type of allocator object that manages memory storage for the 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 multiset.
- All constructors store a function object of type `Compare` that is used to establish an order among the keys of the multiset and that can later be returned by calling `key_comp()`.
- The first three constructors specify an empty initial 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 fourth constructor specifies a copy of the multiset `_Right`.
- The last three constructors copy the range `[_First, _Last)` of a multiset with increasing explicitness in specifying the type of comparison function and allocator.

```
//multiset, constructor
#include <set>
#include <iostream>
using namespace std;

int main()
{
    multiset <int>::iterator mst0_Iter, mst1_Iter, mst2_Iter, mst3_Iter;
    multiset <int>::iterator mst4_Iter, mst5_Iter, mst6_Iter;

    //Create an empty multiset mst0 of key type integer
    multiset <int> mst0;

    //Create an empty multiset mst1 with the key comparison
    //function of less than, then insert 6 elements
    multiset <int, less<int> > mst1;
    mst1.insert(41);
    mst1.insert(15);
    mst1.insert(30);
    mst1.insert(10);
    mst1.insert(15);
    mst1.insert(35);

    //Create an empty multiset mst2 with the key comparison
    //function of greater than, then insert 4 elements
    multiset <int, greater<int> > mst2;
    mst2.insert(10);
    mst2.insert(21);
    mst2.insert(8);
    mst2.insert(21);

    //Create a multiset mst3 with the
    //allocator of multiset mst1
    multiset <int>::allocator_type mst1_Alloc;
    mst1_Alloc = mst1.get_allocator();
    multiset <int> mst3(less<int>(), mst1_Alloc);
    mst3.insert(7);
    mst3.insert(12);
    mst3.insert(9);
}
```



```

//multiset mst4, a copy of multiset mst1
multiset <int> mst4(mst1);

//Create a multiset mst5 by copying the range mst1[_First, _Last)
multiset <int>::const_iterator mst1_PIter, mst1_QIter;
mst1_PIter = mst1.begin();
mst1_QIter = mst1.begin();
mst1_QIter++;
mst1_QIter++;
multiset <int> mst5(mst1_PIter, mst1_QIter);

//Create a multiset mst6 by copying the range mst4[_First, _Last)
//and with the allocator of multiset mst2
multiset <int>::allocator_type mst2_Alloc;
mst2_Alloc = mst2.get_allocator();
multiset <int> mst6(mst4.begin(), ++mst4.begin(), less<int>(), mst2_Alloc);

//-----
cout<<"Operation: multiset <int> mst0\n";
cout<<"mst0 data: ";
for(mst0_Iter = mst0.begin(); mst0_Iter != mst0.end(); mst0_Iter++)
    cout<<" " <<*mst0_Iter;
cout<<endl;

cout<<"\nOperation: multiset <int, less<int> > mst1\n";
cout<<"Operation: mst1.insert(41)... \n";
cout<<"mst1 data: ";
for(mst1_Iter = mst1.begin(); mst1_Iter != mst1.end(); mst1_Iter++)
    cout<<" " <<*mst1_Iter;
cout<<endl;

cout<<"\nOperation: multiset <int, greater<int> > mst2\n";
cout<<"Operation: mst2.insert(10)... \n";
cout<<"mst2 data: "<<*mst2.begin()<<" "<<*++mst2.begin()<<endl;

cout<<"\nOperation: multiset <int> mst3(less<int>(), mst1_Alloc)\n";
cout<<"Operation: mst3.insert(7)... \n";
cout<<"mst3 data: ";
for(mst3_Iter = mst3.begin(); mst3_Iter != mst3.end(); mst3_Iter++)
    cout<<" " <<*mst3_Iter;
cout<<endl;

cout<<"\nOperation: multiset <int> mst4(mst1)\n";
cout<<"mst4 data: ";
for(mst4_Iter = mst4.begin(); mst4_Iter != mst4.end(); mst4_Iter++)
    cout<<" " <<*mst4_Iter;
cout<<endl;

cout<<"\nOperation: multiset <int> mst5(mst1_PIter, mst1_QIter)\n";
cout<<"mst5 data: ";
for(mst5_Iter = mst5.begin(); mst5_Iter != mst5.end(); mst5_Iter++)
    cout<<" " <<*mst5_Iter;
cout<<endl;

cout<<"\nOperation: multiset <int> mst6(mst4.begin(), \n++mst4.begin(), less<int>(),
                                     mst2_Alloc);\n";
cout<<"mst6 data: ";
for(mst6_Iter = mst6.begin(); mst6_Iter != mst6.end(); mst6_Iter++)
    cout<<" " <<*mst6_Iter;
cout<<endl;
return 0;
}

```

**Output:**

```

"e:\projectvc\secure\secure\Debug\secure.exe"
Operation: multiset <int> mst0
mst0 data:

Operation: multiset <int, less<int> > mst1
Operation: mst1.insert(41)...
mst1 data: 10 15 15 30 35 41

Operation: multiset <int, greater<int> > mst2
Operation: mst2.insert(10)...
mst2 data: 21 21

Operation: multiset <int> mst3<less<int>(), mst1_Alloc>
Operation: mst3.insert(7)...
mst3 data: 7 9 12

Operation: multiset <int> mst4<mst1>
mst4 data: 10 15 15 30 35 41

Operation: multiset <int> mst5<mst1_PIter, mst1_QIter>
mst5 data: 10 15

Operation: multiset <int> mst6<mst4.begin(),
+mst4.begin(), less<int>(), mst2_Alloc>;
mst6 data: 10
Press any key to continue

```

### find() program example

- The return value is an iterator or const\_iterator that addresses the first location of an element with a specified key, or the location succeeding the last element in the multiset if no match is found for the key.
- The member function returns an iterator that addresses an element in the multiset whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.
- If the return value of find() is assigned to a const\_iterator, the multiset object cannot be modified. If the return value of find() is assigned to an iterator, the multiset object can be modified.

```

//multiset, find()
#include <set>
#include <iostream>
using namespace std;

int main()
{
    multiset <int> mst1;
    multiset <int>::const_iterator mst1_QIter, mst1_PIter, mst1_RIter;

    mst1.insert(6);
    mst1.insert(2);
    mst1.insert(14);
    mst1.insert(6);
    mst1.insert(10);

    cout<<"mst1 data: ";
    for(mst1_RIter = mst1.begin(); mst1_RIter != mst1.end(); mst1_RIter++)
        cout<<" "<<*mst1_RIter;
    cout<<endl;

    mst1_PIter = mst1.find(10);
    cout<<"The first element of multiset mst1 with a key of 10 is: "
        <<*mst1_PIter<<endl;

    mst1_PIter = mst1.find(21);

    //If no match is found for the key, end() is returned
    if(mst1_PIter == mst1.end())
        cout<<"\n\nThe multiset mst1 doesn't have an element "
            <<"with a key of 21"<<endl;
    else
        cout<<"\n\nThe element of multiset mst1 with a key of 21 is: "
            <<*mst1_PIter<<endl;
}

```

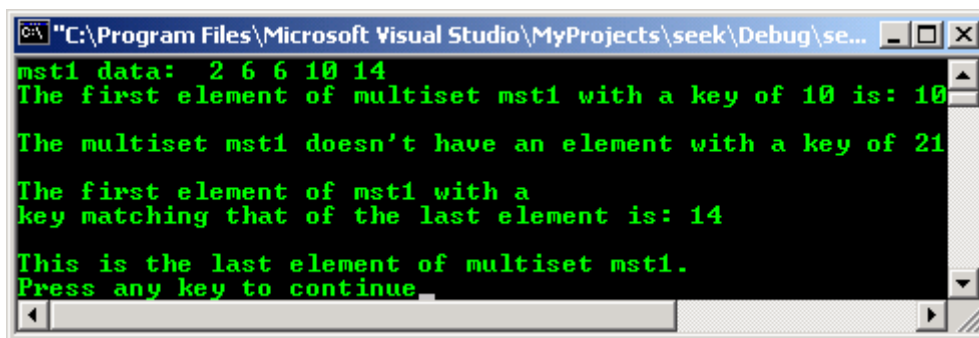
```

//The element at a specific location in the multiset can be
//found using a dereferenced iterator addressing the location
mst1_QIter = mst1.end();
mst1_QIter--;
mst1_PIter = mst1.find(*mst1_QIter);
cout<<"\nThe first element of mst1 with a\key matching "
    <<"that of the last element is: "
    <<*mst1_PIter<<endl;

//Note that the first element with a key equal to
//the key of the last element is not the last element
if(mst1_PIter == --mst1.end())
cout<<"\nThis is the last element of multiset mst1."
    <<endl;
else
cout<<"\nThis is not the last element of multiset mst1."
    <<endl;
return 0;
}

```

**Output :**



- Program examples compiled using **g++**.

```

//*****listsort.cpp*****
//list, sort()
#include <list>
#include <iostream>
using namespace std;

int main()
{
list <int> ls1;
list <int>::iterator ls1Iter;

ls1.push_back(31);
ls1.push_back(12);
ls1.push_back(40);
ls1.push_back(15);
ls1.push_back(9);
ls1.push_back(44);

cout<<"Before sorting, ls1 data: ";
for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
cout<<" "<<*ls1Iter;
cout<<endl;

cout<<"\nOperation: ls1.sort()\n";
ls1.sort();
cout<<"After sorting, ls1 data: ";
for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
cout<<" "<<*ls1Iter;
cout<<endl;

cout<<"\nOperation: ls1.sort(greater<int>())\n";
ls1.sort(greater<int>());
cout<<"Re sort with 'greater than' operation,\nls1 =";
for(ls1Iter = ls1.begin(); ls1Iter != ls1.end(); ls1Iter++)
cout<<" "<<*ls1Iter;
cout<<endl;
return 0;
}

```

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

```
Before sorting, ls1 data: 31 12 40 15 9 44
```

```
Operation: ls1.sort()
After sorting, ls1 data: 9 12 15 31 40 44
```

```
Operation: ls1.sort(greater<int>())
Re sort with 'greater than' operation,
ls1 = 44 40 31 15 12 9
```

```
//*****setcount.cpp*****
//set, count()
//some warning during the compilation
#include <set>
#include <iostream>
using namespace std;

int main()
{
    set <int> st1;
    int i;

    st1.insert(1);
    st1.insert(2);
    st1.insert(1);

    //Keys must be unique in set, duplicates are ignored
    i = st1.count(1);
    cout<<"The number of elements in st1 with a sort key of 1 is: "<<i<<endl;

    i = st1.count(2);
    cout<<"The number of elements in st1 with a sort key of 2 is: "<<i<<endl;

    i = st1.count(3);
    cout<<"The number of elements in st1 with a sort key of 3 is: "<<i<<endl;

    return 0;
}
```

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

```
The number of elements in st1 with a sort key of 1 is: 1
The number of elements in st1 with a sort key of 2 is: 1
The number of elements in st1 with a sort key of 3 is: 0
```

```
//*****multisetfind.cpp*****
//multiset, find()
#include <set>
#include <iostream>
using namespace std;

int main()
{
    multiset <int> mst1;
    multiset <int>::const_iterator mst1_QIter, mst1_PIter, mst1_RIter;

    mst1.insert(6);
    mst1.insert(2);
    mst1.insert(14);
    mst1.insert(6);
    mst1.insert(10);

    cout<<"mst1 data: ";
    for(mst1_RIter = mst1.begin(); mst1_RIter != mst1.end(); mst1_RIter++)
        cout<<" "<<*mst1_RIter;
    cout<<endl;

    mst1_PIter = mst1.find(10);
    cout<<"The first element of multiset mst1 with a key of 10 is: "
         <<*mst1_PIter<<endl;

    mst1_PIter = mst1.find(21);
```

```

//If no match is found for the key, end() is returned
if(mst1_PIter == mst1.end())
cout<<"\nThe multiset mst1 doesn't have an element "
  <<"with a key of 21"<<endl;
else
cout<<"\nThe element of multiset mst1 with a key of 21 is: "
  <<*mst1_PIter<<endl;

//The element at a specific location in the multiset can be
//found using a dereferenced iterator addressing the location
mst1_QIter = mst1.end();
mst1_QIter--;
mst1_PIter = mst1.find(*mst1_QIter);
cout<<"\nThe first element of mst1 with a\nkey matching "
  <<"that of the last element is: "
  <<*mst1_PIter<<endl;

//Note that the first element with a key equal to
//the key of the last element is not the last element
if(mst1_PIter == --mst1.end())
cout<<"\nThis is the last element of multiset mst1."
  <<endl;
else
cout<<"\nThis is not the last element of multiset mst1."
  <<endl;
return 0;
}

```

```
[bodo@bakawali ~]$ g++ multisetfind.cpp -o multisetfind
```

```
[bodo@bakawali ~]$ ./multisetfind
```

```

mst1 data:  2 6 6 10 14
The first element of multiset mst1 with a key of 10 is: 10

The multiset mst1 doesn't have an element with a key of 21

The first element of mst1 with a
key matching that of the last element is: 14

This is the last element of multiset mst1.

```

```

-----End of multiset-----
---www.tenouk.com---

```

### Further reading and digging:

1. Check the [best selling C/C++ and STL books at Amazon.com](#).