MODULE 31
--THE STL--
ITERATOR PART I

My Training Period: hours

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

Abilities

- Able to understand and use iterators.
- Able to understand and use iterator template classes.
- Able to understand and use iterator `typedef`.
- Able to understand and use iterator member functions.
- Able to understand and use iterator operators.
- Able to appreciate how the class and function templates are used.

31.1 Introduction

- In the previous Modules, we have learned how to construct various types of containers. At the same time, in the program examples, iterators and algorithm also have been introduced.
- In this Module we are going to discuss an iterator in more detail.

31.2 Iterators

- An iterator is an object that can iterate or navigate or traverse over elements in the containers that represent data structures. These elements may be the entire or just a portion of a STL container. It represents a certain position in a container.
- For example, the following basic operations: **output**, **input**, **forward**, **bidirectional** and **random access** define the behavior of an iterator.
- Iterators are a generalization of pointers, abstracting from their requirements in a way that allows a C++ program to work with different data structures in a uniform manner. Iterators act as intermediaries between containers and generic algorithms.
- Instead of operating on specific data types, algorithms are defined to operate on a range specified by a type of iterator. Any data structure that satisfies the requirements of the iterator may then be operated on by the algorithm.
- The name of an iterator type or its prefix indicates the category of iterators required for that type.
- There are five types or categories of iterator, each with its own set of requirements and operations are shown below.
- They are arranged in the order of the strength of their functionalities.

Iterator Type	Description
Output	Forward moving, may store but not retrieve values, provided by `ostream` and `inserter`. An output iterator *I* can only have a value *V* stored indirect on it, after which it must be incremented before the next store, as in (**I* ++ = *V*), (**I* = *V*, ++ *I*), or (**I* = *V*, *I* ++).
Input	Forward moving, may retrieve but not store values, provided by `istream`. An input iterator *I* can represent a singular value that indicates end of sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value *V* accessed indirect on it any number of times, as in (*V* = **I*). To progress to the next value or end of sequence, you increment it, as in ++ *I*, *I* ++, or (*V* = **I* ++). Once you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
Forward	Forward moving, may store and retrieve values. A forward iterator *I* can take the place of an output iterator for writing or an input iterator for reading. You can, however, read (through *V* = **I*) what you just wrote (through **I* = *V*) through a forward iterator. You can also make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
Bidirectional	Forward and backward moving, may store and retrieve values, provided by list, set, multiset, map, and multimap. A bidirectional iterator *X* can take the

	place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in -- *I*, *I* --, or (*V* = **I* --).
Random-access	Elements accessed in any order, may store and retrieve values, provided by vector, deque, string, and array. A random-access iterator *I* can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For *N*, an integer object, you can write *x*[*N*], *x* + *N*, *x* - *N*, and *N* + *X*.

Table 31.1: Iterator types

- Note that an object pointer can take the place of a random-access iterator or any other iterator. All iterators can be assigned or copied.
- They are assumed to be lightweight objects and are often passed and returned by value, not by reference. Note also that none of the operations previously described can throw an exception when performed on a valid iterator.
- The hierarchy of iterator categories can be summarized by showing three sequences. For **write-only access** to a sequence, you can use any of the following:

```
Output iterator          or can be replaced by
forward iterator         or can be replaced by
bidirectional iterator   or can be replaced by
random-access iterator
```

- Any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but not the other way around.
- For **read-only access** to a sequence, you can use any of the following:

```
Input iterator           or can be replaced by
forward iterator         or can be replaced by
bidirectional iterator   or can be replaced by
random-access iterator
```

- An input iterator is the weakest of all categories, in this case.
- Finally, for **read/write access** to a sequence, you can use any of the following:

```
forward iterator         or can be replaced by
bidirectional iterator   or can be replaced by
random-access iterator
```

- An object pointer can always serve as a random-access iterator, so it can serve as any category of iterator if it supports the proper read/write access to the sequence it designates.
- An iterator `Iterator` other than an object pointer must also define the member types required by the specialization `iterator_traits<Iterator>`. Note that these requirements can be met by deriving `Iterator` from the public base class iterator.
- It is important to understand the promises and limitations of each iterator category to see how iterators are used by containers and algorithms in the STL.
- For simple example:

Operator	Description
`++`	Make the iterator step forward to the next element.
`==` and `!=`	Return whether two iterators represent the same position or not.
`=`	Assigns an iterator.

Table 31.2: Operators and iterator

- Compared to the traditional usage of this operator on arrays, iterators are smart pointers that iterate over more complicated data structures of containers.
- Each container type supplies its own kind of iterator.
- Hence, iterators share the same interface but have different types, then operations use the same interface but different types, and we can use templates to formulate generic operations that work with arbitrary types that satisfy the interface.

- All container classes provide the same basic member functions that enable them to use iterators to navigate over their elements.
- The most frequently functions used in the program examples in the previous Modules are `begin()` and `end()`.

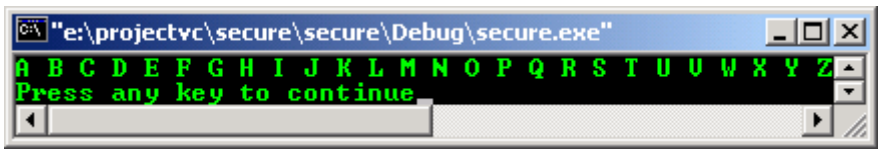
Member function	Description
`begin()`	Returns an iterator that represents the beginning of the elements in the container. The beginning is the position of the first element (if any).
`end()`	Returns an iterator that represents the end of the elements in the container. As shown in the previous modules before, the end is the position behind the last element.

Table 31.3: `begin()` and `end()` member functions

- The following example demonstrates the simple use of iterators.

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

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

//append elements from 'A' to 'Z'
//to the list lst container
for(char chs='A'; chs<='Z'; ++chs)
lst.push_back(chs);

//iterate over all elements and print,
//separated by space
list<char>::const_iterator pos;
for(pos = lst.begin(); pos != lst.end(); ++pos)
cout<<*pos<<' ';
cout<<endl;
return 0;
}
```

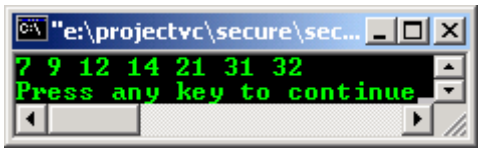
Output:




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

int main()
{
//set container of int
//data type
set<int> tst;

//insert elements
tst.insert(12);
tst.insert(21);
tst.insert(32);
tst.insert(31);
tst.insert(9);
tst.insert(14);
tst.insert(21);
tst.insert(31);
tst.insert(7);

//iterate over all elements and print,
//separated by space
```

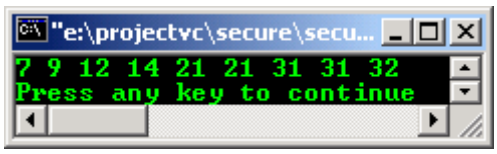
```
set<int>::const_iterator pos;
//preincrement and predecrement are faster
//than postincrement and postdecrement...
for(pos = tst.begin(); pos != tst.end(); ++pos)
cout<<*pos<<' ';
cout<<endl;
return 0;
}
```

Output:




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

int main()
{
//multiset container of int
//data type
multiset<int> tst;

//insert elements
tst.insert(12);
tst.insert(21);
tst.insert(32);
tst.insert(31);
tst.insert(9);
tst.insert(14);
tst.insert(21);
tst.insert(31);
tst.insert(7);

//iterate over all elements and print,
//separated by space
multiset<int>::const_iterator pos;
//preincrement and predecrement are faster
//than postincrement and postdecrement...
for(pos = tst.begin(); pos != tst.end(); ++pos)
cout<<*pos<<' ';
cout<<endl;
return 0;
}
```

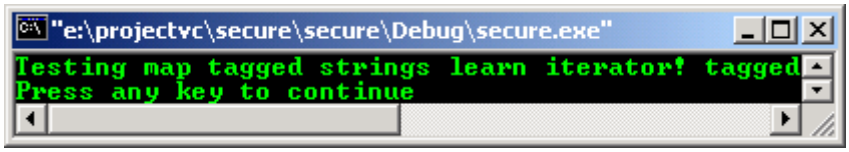
Output:




```
//iterator, map simple example
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
//type of the collection
map<int, string> mp;

//set container for int/string values
//insert some elements in arbitrary order
//notice a value with key 1...
mp.insert(make_pair(5,"learn"));
mp.insert(make_pair(2,"map"));
mp.insert(make_pair(1,"Testing"));
mp.insert(make_pair(7,"tagged"));
```

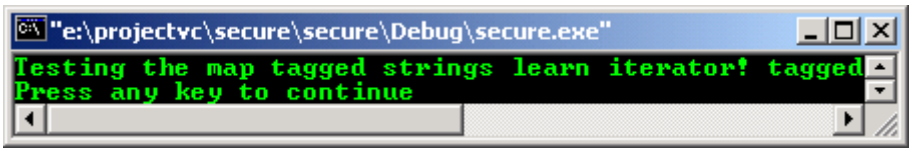
```
mp.insert(make_pair(4,"strings"));
mp.insert(make_pair(6,"iterator!"));
mp.insert(make_pair(1,"the"));
mp.insert(make_pair(3,"tagged"));

//iterate over all elements and print,
//element member second is the value
map<int, string>::iterator pos;
for(pos = mp.begin(); pos != mp.end(); ++pos)
cout<<pos->second<<' ';
cout<<endl;
return 0;
}
```

Output:




```
//iterator, multimap simple example
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
//type of the collection
multimap<int, string> mmp;

//set container for int/string values
//insert some elements in arbitrary order
//notice a value of key 1
mmp.insert(make_pair(5,"learn"));
mmp.insert(make_pair(2,"map"));
mmp.insert(make_pair(1,"Testing"));
mmp.insert(make_pair(7,"tagged"));
mmp.insert(make_pair(4,"strings"));
mmp.insert(make_pair(6,"iterator!"));
mmp.insert(make_pair(1,"the"));
mmp.insert(make_pair(3,"tagged"));

//iterate over all elements and print,
//element member second is the value
multimap<int, string>::iterator pos;
for(pos = mmp.begin(); pos != mmp.end(); ++pos)
cout<<pos->second<<' ';
cout<<endl;
return 0;
}
```

Output:



31.3 Iterator Categories

- Iterators are subdivided into different categories that are based on their general abilities. The iterators of the predefined container classes belong to one of the following two categories:

Category	Description
Bidirectional iterator	Bidirectional iterators are able to iterate in two directions, forward and backward, by using the increment operator and decrement operators respectively. The iterators of the container classes list, set, multiset, map, and multimap are bidirectional iterators.
Random access iterator	Random access iterators have all the properties of bidirectional iterators

	plus they can perform random access. You can add and subtract offsets, process differences, and compare iterators by using relational operators such as < and >. The iterators of the container classes' vector and deque, and iterators of strings are random access iterators.

Table 31.4: Iterator category

- We should not use special operations for random access iterators in order to write generic code that is as independent of the container type as possible. For example, the following loop works with any container:

```
for(pos = contner.begin(); pos != contner.end(); ++pos)
{...}
```

- However, the following does not work with all containers:

```
for(pos = contner.begin(); pos < contner.end(); ++pos)
{...}
```

- Operator < is only provided for random access iterators, so this loop does not work with lists, sets, and maps. To write generic code for arbitrary containers, you should use `operator !=` rather than `operator <`.
- A category only defines the abilities of iterators, not the type of the iterators.
- Let dig more details what are provided for us in <`iterator`> header.

31.4 <`iterator`> Header

- Defines the iterator primitives, predefined iterators and stream iterators, as well as several supporting templates. The predefined iterators include `insert` and `reverse` adaptors.
- There are three classes of insert iterator adaptors: **front**, **back**, and **general**.
- They provide insert semantics rather than the overwrite semantics that the container member function iterators provide. To use iterator we must include the iterator header as shown below.

```
#include <iterator>
```

<`iterator`> Header Members

Member Functions

Member function	Description
`advance()`	Increments an iterator by a specified number of positions.
`back_inserter()`	Creates an iterator that can insert elements at the back of a specified container.
`distance()`	Determines the number of increments between the positions addressed by two iterators.
`front_inserter()`	Creates an iterator that can insert elements at the front of a specified container.
`inserter()`	An iterator adaptor that adds a new element to a container at a specified point of insertion.

Table 31.5: Iterator member functions

`advance()`

```
template<class InputIterator, class Distance>
   void advance(
      InputIterator& _InIt,
      Distance _Off
   );
```

Parameters

Parameter	Description
`_InIt`	The iterator that is to be incremented and that must satisfy the requirements for an input iterator.

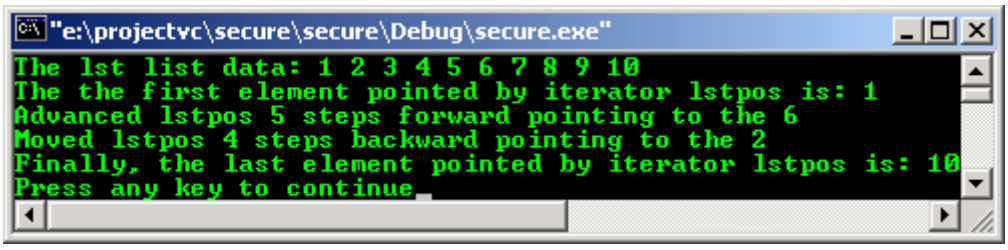
_Off	An integral type that is convertible to the iterator's difference type and that specifies the number of increments the position of the iterator is to be advanced.

Table 31.6

- The range advanced through must be nonsingular, where the iterators must be dereferenceable or past the end.
- If the `InputIterator` satisfies the requirements for a bidirectional iterator type, then `_Off` may be negative. If `InputIterator` is an input or forward iterator type, `_Off` must be nonnegative.
- The advance function has constant complexity when `InputIterator` satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

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

int main()
{
int i;

list<int> lst;
for(i = 1; i <= 10; ++i)
lst.push_back(i);

list<int>::iterator lstIter, lstpos = lst.begin();

cout<<"The lst list data: ";
for(lstIter = lst.begin(); lstIter != lst.end(); lstIter++)
cout<<*lstIter<<" ";
cout<<endl;
cout<<"The the first element pointed by iterator lstpos is: "<<*lstpos<<endl;

advance(lstpos, 5);
cout<<"Advanced lstpos 5 steps forward pointing to the "<<*lstpos<<endl;

advance(lstpos, -4);
cout<<"Moved lstpos 4 steps backward pointing to the "<<*lstpos<<endl;

advance(lstpos, 8);
cout<<"Finally, the last element pointed by iterator lstpos is: "<<*lstpos<<endl;

return 0;
}
```

Output:



back_inserter()

```
template<class Container>
   back_insert_iterator<Container> back_inserter(
      Container& _Cont
   );
```

Parameter

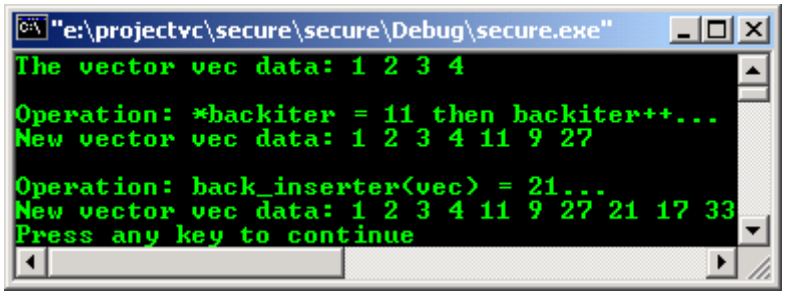
Parameter	Description
_Cont	The container into which the back insertion is to be executed.

Table 31.7

- The return value is a `back_insert_iterator` associated with the container object `_Cont`.
- Within the Standard Template Library, the argument must refer to one of the three sequence containers that have the member function `push_back()`: deque Class, list Class, or vector Class.

```
//iterator, back_inserter()
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;
vector<int> vec;
for(i = 1; i < 5; ++i)
vec.push_back(i);

vector <int>::iterator vecIter;
cout<<"The vector vec data: ";
for(vecIter = vec.begin(); vecIter != vec.end(); vecIter++)
cout<<*vecIter<<" ";
cout<<endl;

//Insertions using template function
back_insert_iterator<vector<int> > backiter(vec);
*backiter = 11;
backiter++;
*backiter = 9;
backiter++;
*backiter = 27;

cout<<"\nOperation: *backiter = 11 then backiter++...\n";
cout<<"New vector vec data: ";
for(vecIter = vec.begin(); vecIter != vec.end(); vecIter++)
cout<<*vecIter<<" ";
cout<<endl;

cout<<"\nOperation: back_inserter(vec) = 21...\n";
//Alternatively, insertions using the
//back_inserter() member function
back_inserter(vec) = 21;
back_inserter(vec) = 17;
back_inserter(vec) = 33;
cout<<"New vector vec data: ";
for(vecIter = vec.begin(); vecIter != vec.end(); vecIter++)
cout<<*vecIter<<" ";
cout<<endl;
return 0;
}
```

Output:



distance()

```
template<class InputIterator>
   typename iterator_traits<InputIterator>::difference_type
      distance(
         InputIterator _First,
         InputIterator _Last
      );
```

Parameters

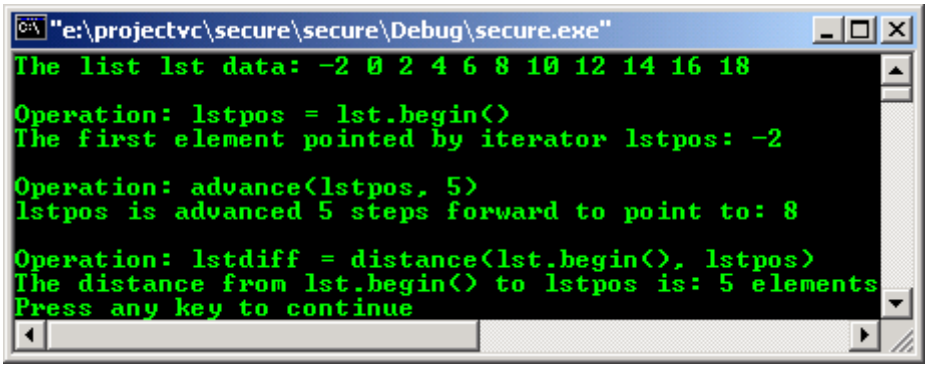
Parameter	Description
_First	The first iterator whose distance from the second is to be determined.
_Last	The second iterator whose distance from the first is to be determined.

Table 31.8

- The return value is the number of times that _First must be incremented until it equal _Last.
- The advance function has constant complexity when InputIterator satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

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

int main()
{
int i;

list<int> lst;
for(i = -1; i < 10; ++i)
lst.push_back(2*i);

list <int>::iterator lstiter, lstpos = lst.begin();
cout<<"The list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

cout<<"\nOperation: lstpos = lst.begin()\n";
cout<<"The first element pointed by iterator lstpos: "<<*lstpos<<endl;

cout<<"\nOperation: advance(lstpos, 5)\n";
advance(lstpos, 5);
cout<<"lstpos is advanced 5 steps forward to point to: "<<*lstpos<<endl;

list<int>::difference_type lstdiff;
cout<<"\nOperation: lstdiff = distance(lst.begin(), lstpos)\n";
lstdiff = distance(lst.begin(), lstpos);
cout<<"The distance from lst.begin() to lstpos is: "<<lstdiff<<" elements"<<endl;
return 0;
}
```

Output



front_inserter()

```
template<class Container>
   front_insert_iterator<Container> front_inserter(
      Container& _Cont
   );
```

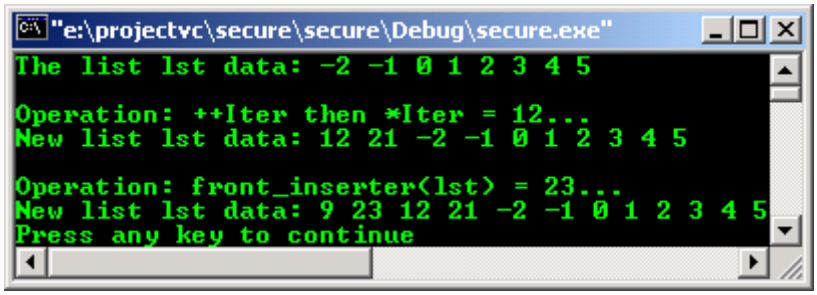
Parameter

Parameter	Description
_Cont	The container object whose front is having an element inserted.

- The return value is a `front_insert_iterator()` associated with the container object `_Cont`.
- The member function `front_insert_iterator()` of the `front_insert_iterator` class may also be used.
- Within the STL, the argument must refer to one of the two sequence containers that have the member function `push_back()`: deque Class or list Class.

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

int main()
{
int i;
list<int>::iterator lstiter;

list<int> lst;
for(i = -2; i<=5; ++i)
lst.push_back(i);

cout<<"The list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Using the template function to insert an element
cout<<"\nOperation: ++Iter then *Iter = 12...\n";
front_insert_iterator< list < int> > Iter(lst);
*Iter = 21;
++Iter;
*Iter = 12;

cout<<"New list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

cout<<"\nOperation: front_inserter(lst) = 23...\n";
//Alternatively, using the front_insert() member function
front_inserter(lst) = 23;
front_inserter(lst) = 9;

cout<<"New list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;
return 0;
}
```

Output:



inserter()

```
template<class Container, class Iterator>
   insert_iterator<Container> inserter(
      Container& _Cont,
      Iterator _It
   );
```

Parameters

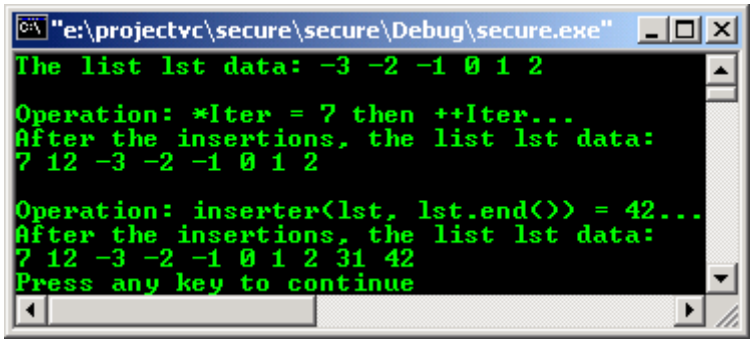
Parameter	Description
_Cont	The container to which new elements are to be added.
_It	An iterator locating the point of insertion.

Table 31.10

- The return value is an insert iterator addressing the new element inserted.
- Within the STL, the sequence and sorted associative containers may be used as a container object _Cont with the templatized inserter.

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

int main()
{
int i;
list <int>::iterator lstiter;

list<int> lst;
for(i = -3; i<=2; ++i)
lst.push_back(i);

cout<<"The list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Using the template version to insert an element
insert_iterator<list <int> > Iter(lst, lst.begin());
*Iter = 7;
++Iter;
*Iter = 12;

cout<<"\nOperation: *Iter = 7 then ++Iter...\n";
cout<<"After the insertions, the list lst data: \n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Alternatively, using the member function inserter()
//to insert an element
inserter(lst, lst.end()) = 31;
inserter(lst, lst.end()) = 42;

cout<<"\nOperation: inserter(lst, lst.end()) = 42...\n";
cout<<"After the insertions, the list lst data: \n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;
return 0;
}
```

Output:



Operators

Operator	Description
operator!=	Tests if the iterator object on the left side of the operator is not equal to the iterator object on the right side.
operator==	Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.
operator<	Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.
operator<=	Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.
operator>	Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.
operator>=	Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.
operator+	Adds an offset to an iterator and returns the new reverse_iterator addressing the inserted element at the new offset position.
operator-	Subtracts one iterator from another and returns the difference.

Table 31.11

operator!=

```
template<class RandomIterator>
   bool operator!=(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
template<class Type, class CharType, class Traits, class Distance>
   bool operator!=(
      const istream_iterator<Type, CharType, Traits, Distance>& _Left,
      const istream_iterator<Type, CharType, Traits, Distance>& _Right
   );
template<class CharType, class Tr>
   bool operator!=(
      const istreambuf_iterator<CharType, Traits>& _Left,
      const istreambuf_iterator<CharType, Traits>& _Right
);
```

Parameters

Parameter	Description
_Left	An object of type iterator.
_Right	An object of type iterator.

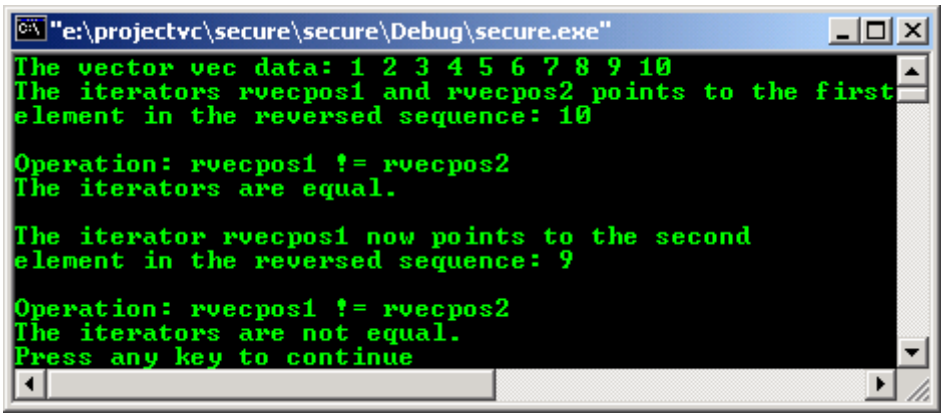
Table 31.12

- The return value is **true** if the iterator objects are not equal; **false** if the iterator objects are equal.
- One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

```
//iterator, operator!=
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;
vector<int> vec;
for(i = 1; i<=10; ++i)
vec.push_back(i);

vector<int>::iterator veciter;
cout<<"The vector vec data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

//Initializing reverse_iterators to the last element
vector<int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"The iterators rvecpos1 and rvecpos2 points to the first\n"
```

```
<<"element in the reversed sequence: "<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos1 != rvecpos2\n";
if(rvecpos1 != rvecpos2)
cout<<"The iterators are not equal."<<endl;
else
cout<<"The iterators are equal."<<endl;

rvecpos1++;
cout<<"\nThe iterator rvecpos1 now points to the second\n"
<<"element in the reversed sequence: "<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos1 != rvecpos2\n";
if(rvecpos1 != rvecpos2)
cout<<"The iterators are not equal."<<endl;
else
cout<<"The iterators are equal."<<endl;
return 0;
}
```

Output:



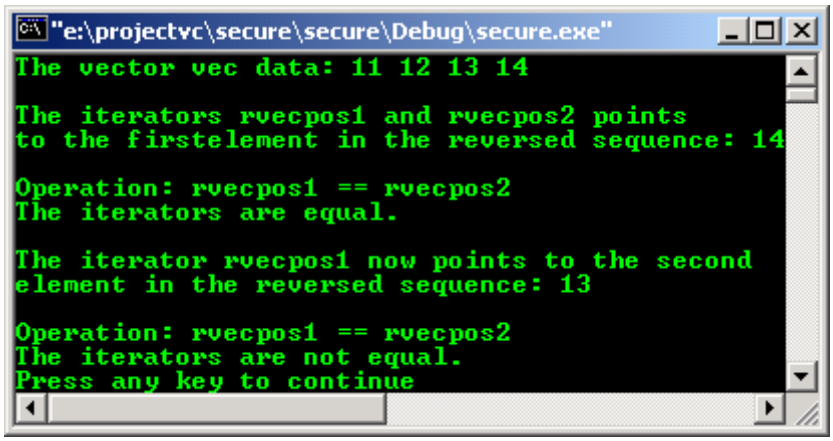
operator==

- The return value is **true** if the iterator objects are equal; **false** if the iterator objects are not equal.
- One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they are not equal.

```
//iterator, operator==
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;

vector<int> vec;
for(i = 11; i<15; ++i)
vec.push_back(i);

vector <int>::iterator veciter;
cout<<"The vector vec data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

//Initializing reverse_iterators to the last element
vector<int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"\nThe iterators rvecpos1 and rvecpos2 points\nto the first"
<<"element in the reversed sequence: "<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos1 == rvecpos2\n";
if(rvecpos1 == rvecpos2)
cout<<"The iterators are equal."<<endl;
else
cout<<"The iterators are not equal."<<endl;
```

```
rvecpos1++;
cout<<"\nThe iterator rvecpos1 now points to the second\n"
<<"element in the reversed sequence: "<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos1 == rvecpos2\n";
if(rvecpos1 == rvecpos2)
cout<<"The iterators are equal."<<endl;
else
cout<<"The iterators are not equal."<<endl;
return 0;
}
```

Output:



operator<

```
template<class RandomIterator>
   bool operator<(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

Parameter	Description
_Left	An object of type iterator.
_Right	An object of type iterator.

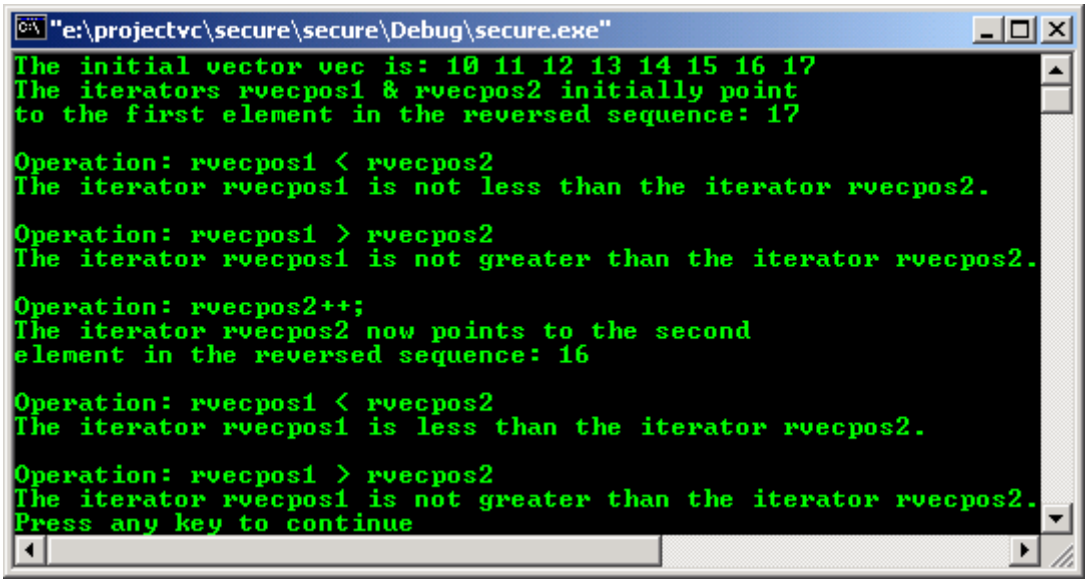
Table 31.13

- The return value is **true** if the iterator on the left side of the expression is less than the iterator on the right side of the expression; **false** if it is greater than or equal to the iterator on the right.
- One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object.
- One iterator object is not less than another if it addresses either the same element as the other iterator object or an element that occurs later in the container than the element addressed by the other iterator object.

```
//iterator, operator<
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
using namespace std;
int i;
vector<int> vec;
for(i = 10; i<= 17; ++i)
vec.push_back(i);

vector<int>::iterator veciter;
cout<<"The initial vector vec is: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
```

```
cout<<*veciter<<" ";
cout<<endl;

//Initializing reverse_iterators to the last element
vector<int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"The iterators rvecpos1 & rvecpos2 initially point\nto the "
<<"first element in the reversed sequence: "
<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos1 < rvecpos2\n";
if(rvecpos1 < rvecpos2)
cout<<"The iterator rvecpos1 is less than"
<<" the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is not less than"
<<" the iterator rvecpos2."<<endl;

cout<<"\nOperation: rvecpos1 > rvecpos2\n";
if(rvecpos1 > rvecpos2)
cout<<"The iterator rvecpos1 is greater than"
<<" the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is not greater than"
<<" the iterator rvecpos2."<<endl;

cout<<"\nOperation: rvecpos2++;\n";
rvecpos2++;
cout<<"The iterator rvecpos2 now points to the second\n"
<<"element in the reversed sequence: "
<<*rvecpos2<<endl;

cout<<"\nOperation: rvecpos1 < rvecpos2\n";
if(rvecpos1 < rvecpos2)
cout<<"The iterator rvecpos1 is less than"
<<" the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is not less than"
<<" the iterator rvecpos2."<<endl;

cout<<"\nOperation: rvecpos1 > rvecpos2\n";
if(rvecpos1 > rvecpos2)
cout<<"The iterator rvecpos1 is greater than"
<<" the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is not greater than"
<<" the iterator rvecpos2."<<endl;
return 0;
}
```

Output:

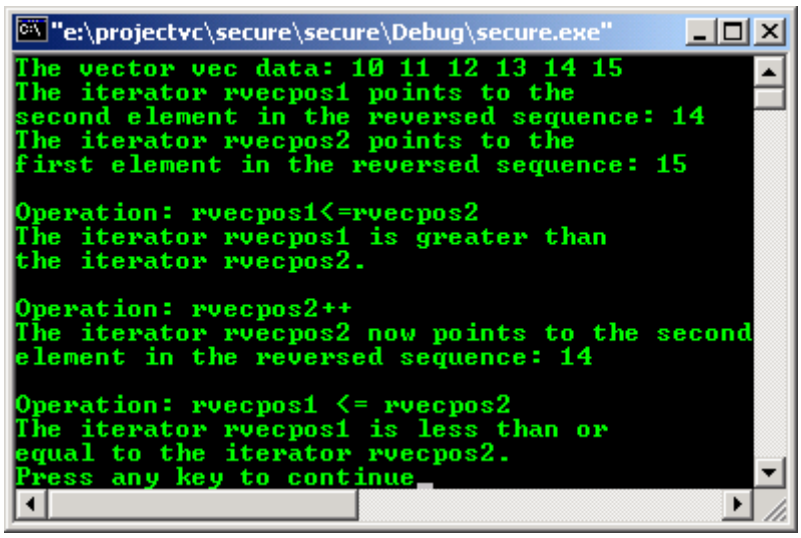


operator<=

- The return value is **true** if the iterator on the left side of the expression is less than or equal to the iterator on the right side of the expression; **false** if it is greater than the iterator on the right.
- One iterator object is less than or equal to another if it addresses the same element or an element that occurs earlier in the container than the element addressed by the other iterator object.
- One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object.

```
//iterator, operator<=
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;
vector<int> vec;
for(i = 10; i<= 15; ++i)
vec.push_back(i);

vector <int>::iterator veciter;
cout<<"The vector vec data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

vector <int>::reverse_iterator rvecpos1 = vec.rbegin()+1, rvecpos2 = vec.rbegin();

cout<<"The iterator rvecpos1 points to the\n"
<<"second element in the reversed sequence: "
<<*rvecpos1<<endl;

cout<<"The iterator rvecpos2 points to the\n"
<<"first element in the reversed sequence: "
<<*rvecpos2<<endl;

cout<<"\nOperation: rvecpos1<=rvecpos2\n";
if(rvecpos1<=rvecpos2)
cout<<"The iterator rvecpos1 is less than or\n"
<<"equal to the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is greater than\n"
<<"the iterator rvecpos2."<<endl;

cout<<"\nOperation: rvecpos2++\n";
rvecpos2++;
cout<<"The iterator rvecpos2 now points to the second\n"
<<"element in the reversed sequence: "<<*rvecpos2<<endl;

cout<<"\nOperation: rvecpos1 <= rvecpos2\n";
if(rvecpos1 <= rvecpos2)
cout<<"The iterator rvecpos1 is less than or\n"
<<"equal to the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is greater than\n"
<<"the iterator rvecpos2."<<endl;
return 0;
}
```

Output:



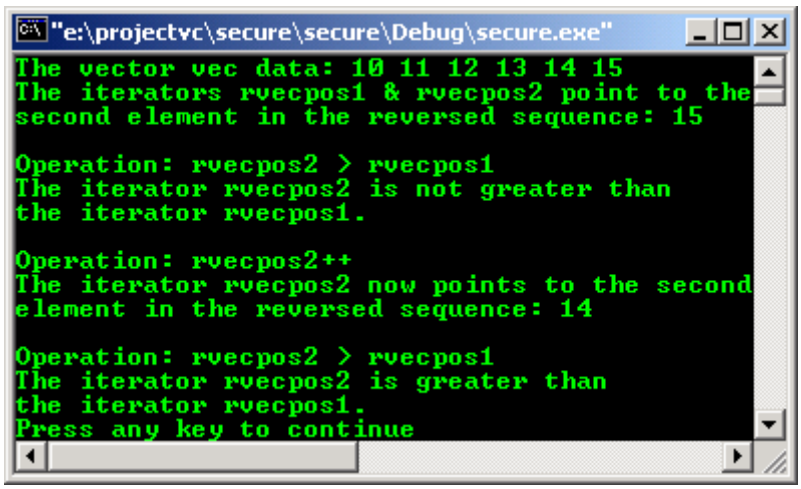
operator>

- The return value is **true** if the iterator on the left side of the expression is greater than the iterator on the right side of the expression; **false** if it is less than or equal to the iterator on the right.
- One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object.
- One iterator object is not greater than another if it addresses either the same element as the other iterator object or an element that occurs earlier in the container than the element addressed by the other iterator object.

```
//iterator, operator<=
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;
vector<int> vec;
for(i = 10; i<= 15; ++i)
vec.push_back(i);

vector <int>::iterator veciter;
cout<<"The vector vec data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

vector <int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"The iterators rvecpos1 & rvecpos2 point to the\n"
<<"second element in the reversed sequence: "
<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos2 > rvecpos1\n";
if(rvecpos2 > rvecpos1)
cout<<"The iterator rvecpos2 is greater than\n"
<<"the iterator rvecpos1."<<endl;
else
cout<<"The iterator rvecpos2 is not greater than\n"
<<"the iterator rvecpos1."<<endl;

cout<<"\nOperation: rvecpos2++\n";
rvecpos2++;
cout<<"The iterator rvecpos2 now points to the second\n"
<<"element in the reversed sequence: "<<*rvecpos2<<endl;

cout<<"\nOperation: rvecpos2 > rvecpos1\n";
if(rvecpos2 > rvecpos1)
cout<<"The iterator rvecpos2 is greater than\n"
<<"the iterator rvecpos1."<<endl;
else
cout<<"The iterator rvecpos2 is not greater than\n"
```

```
<<"the iterator rvecpos1."<<endl;
return 0;
}
```

Output:



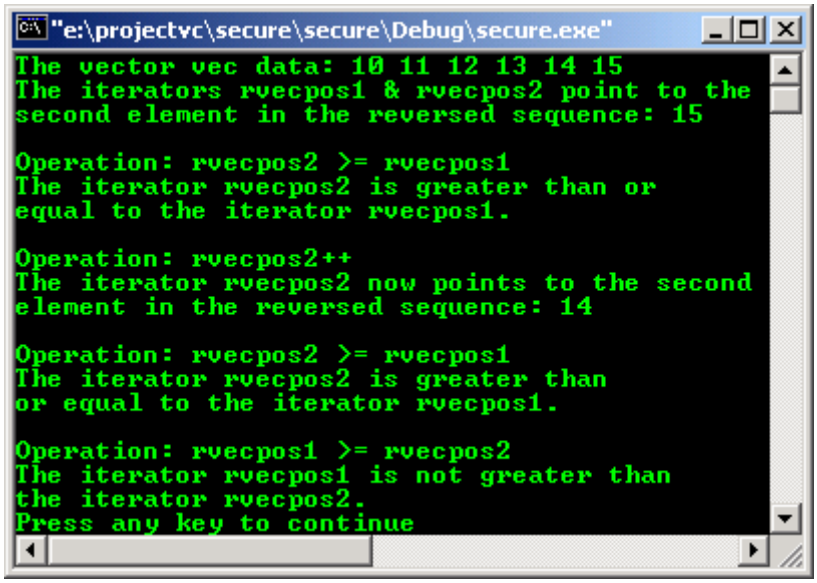
operator>=

- The return value is **true** if the iterator on the left side of the expression is greater than or equal to the iterator on the right side of the expression; **false** if it is less than the iterator on the right.
- One iterator object is greater than or equal to another if it addresses the same element or an element that occurs later in the container than the element addressed by the other iterator object.
- One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object.

```
//iterator, operator>=
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;
vector<int> vec;
for(i = 10; i<= 15; ++i)
vec.push_back(i);

vector <int>::iterator veciter;
cout<<"The vector vec data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

vector <int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"The iterators rvecpos1 & rvecpos2 point to the\n"
<<"second element in the reversed sequence: "
<<*rvecpos1<<endl;

cout<<"\nOperation: rvecpos2 >= rvecpos1\n";
if(rvecpos2 >= rvecpos1)
cout<<"The iterator rvecpos2 is greater than or\n"
<<"equal to the iterator rvecpos1."<<endl;
else
cout<<"The iterator rvecpos2 is not greater than\n"
<<"the iterator rvecpos1."<<endl;

cout<<"\nOperation: rvecpos2++\n";
rvecpos2++;
cout<<"The iterator rvecpos2 now points to the second\n"
<<"element in the reversed sequence: "<<*rvecpos2<<endl;

cout<<"\nOperation: rvecpos2 >= rvecpos1\n";
if(rvecpos2 >= rvecpos1)
cout<<"The iterator rvecpos2 is greater than\n"
```

```
<<"or equal to the iterator rvecpos1."<<endl;
else
cout<<"The iterator rvecpos2 is not greater than\n"
<<"the iterator rvecpos1."<<endl;

cout<<"\nOperation: rvecpos1 >= rvecpos2\n";
if(rvecpos1 >= rvecpos2)
cout<<"The iterator rvecpos1 is greater than\n"
<<"or equal to the iterator rvecpos2."<<endl;
else
cout<<"The iterator rvecpos1 is not greater than\n"
<<"the iterator rvecpos2."<<endl;
return 0;
}
```

Output:



operator+

```
template<class RandomIterator>
   reverse_iterator<RandomIterator> operator+(
      typename reverse_iterator<Iterator>::difference_type _Off,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

Parameter	Description
_Off	The number of positions the const reverse_iterator is to be offset.
_Right	The const reverse_iterator that is to be offset.

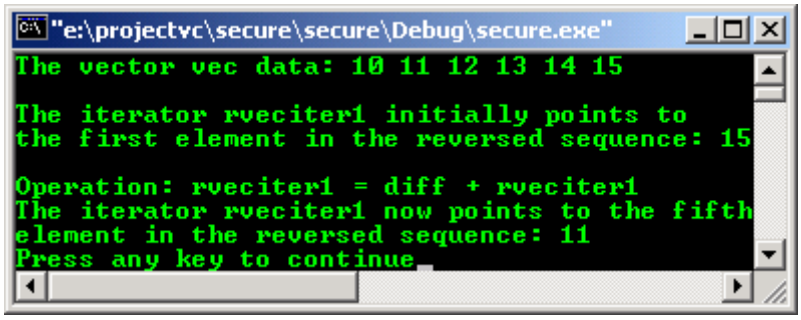
Table 31.14

- The return value is the `sum _Right + _Off`.

```
//iterator, operator+
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;

vector<int> vec;
for(i = 10; i<=15; ++i)
vec.push_back(i);

vector<int>::iterator veciter;
cout<<"The vector vec data: ";
```

```
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

vector<int>::reverse_iterator rveciter1 = vec.rbegin();
cout<<"\nThe iterator rveciter1 initially points to\n"
<<"the first element in the reversed sequence: "
<<*rveciter1<<endl;

cout<<"\nOperation: rveciter1 = diff + rveciter1\n";
vector<int>::difference_type diff = 4;
rveciter1 = diff + rveciter1;
cout<<"The iterator rveciter1 now points to the fifth\n"
<<"element in the reversed sequence: "
<<*rveciter1<<endl;

return 0;
}
```

Output:



operator-

```
template<class RandomIterator>
   typename reverse_iterator<RandomIterator>::difference_type operator-(
      const reverse_iterator<RandomIterator>& _Left,
      const reverse_iterator<RandomIterator>& _Right
   );
```

Parameters

Parameter	Description
_Left	An iterator that serves as the minuend from which another iterator is to be subtracted in forming the difference.
_Right	An iterator that serves as the subtrahend that is to be subtracted from other iterator in forming the difference.

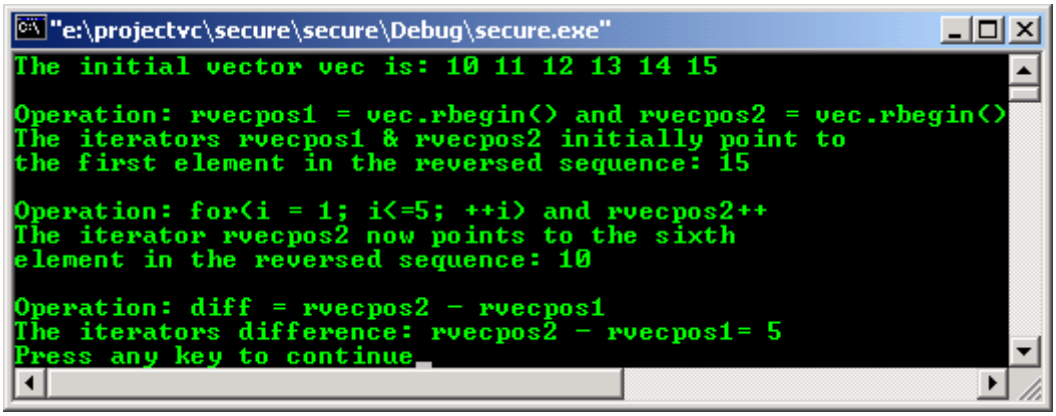
Table 31.15

- The return value is the difference between two iterators: `_Left - _Right`. The difference equals the minuend minus the subtrahend.

```
//iterator, operator-
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;

vector<int> vec;
for(i = 10; i<=15; ++i)
vec.push_back(i);

vector<int>::iterator veciter;
cout<<"The initial vector vec is: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;
```

```
cout<<"\nOperation: rvecpos1 = vec.rbegin() and rvecpos2 = vec.rbegin()\n";
vector<int>::reverse_iterator rvecpos1 = vec.rbegin(), rvecpos2 = vec.rbegin();

cout<<"The iterators rvecpos1 & rvecpos2 initially point to\n"
<<"the first element in the reversed sequence: "<<*rvecpos1<<endl;

cout<<"\nOperation: for(i = 1; i<=5; ++i) and rvecpos2++\n";
for(i = 1; i<=5; ++i)
rvecpos2++;

cout<<"The iterator rvecpos2 now points to the sixth\n"
<<"element in the reversed sequence: "<<*rvecpos2<<endl;

cout<<"\nOperation: diff = rvecpos2 - rvecpos1\n";
vector<int>::difference_type diff = rvecpos2 - rvecpos1;
cout<<"The iterators difference: rvecpos2 - rvecpos1= "<<diff<<endl;
return 0;
}
```

Output:



Iterator Template Classes

Class	Description
`back_insert_iterator`	The template class describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container.
`bidirectional_iterator_tag`	A class that provides a return type for an `iterator_category()` function that represents a bidirectional iterator.
`front_insert_iterator`	The template class describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container.
`forward_iterator_tag`	A class that provides a return type for an `iterator_category()` function that represents a forward iterator.
`input_iterator_tag`	A class that provides a return type for an `iterator_category()` function that represents a bidirectional iterator.
`insert_iterator`	The template class describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container. It also stores the protected `iterator` object, of class `Container::iterator`, called `iter`.
`istream_iterator`	The template class describes an input iterator object. It extracts objects of class `Ty` from an input stream, which it accesses through an object it stores, of type pointer to `basic_istream<Elem, Tr>`.
`istreambuf_iterator`	The template class describes an output iterator object. It inserts elements of class `Elem` into an output stream buffer, which it accesses through an object it stores, of type `pointer` to `basic_streambuf<Elem, Tr>`.
`iterator`	The template class is used as a base type for all iterators.
`iterator_traits`	A template helper class providing critical types that are associated with different iterator types so that they can be referred to in the same way.
`ostream_iterator`	The template class describes an output iterator object. It inserts objects of class `Type` into an output stream, which it accesses through an object it

	stores, of type `pointer` to `basic_ostream<Elem, Tr>`.
`ostreambuf_iterator Class`	The template class describes an output iterator object. It inserts elements of class `Elem` into an output stream buffer, which it accesses through an object it stores, of type pointer to `basic_streambuf<Elem, Tr>`.
`output_iterator_tag`	A class that provides a return type for `iterator_category()` function that represents an output iterator.
`random_access_iterator_tag`	A class that provides a return type for `iterator_category()` function that represents a random-access iterator.
`reverse_iterator`	The template class describes an object that behaves like a random-access iterator, only in reverse.

Table 31.16

31.5 Some Of The Iterator Template Classes program examples

`random_access_iterator_tag` Template Class

- A class that provides a return type for `iterator_category` function that represents a random-access iterator.

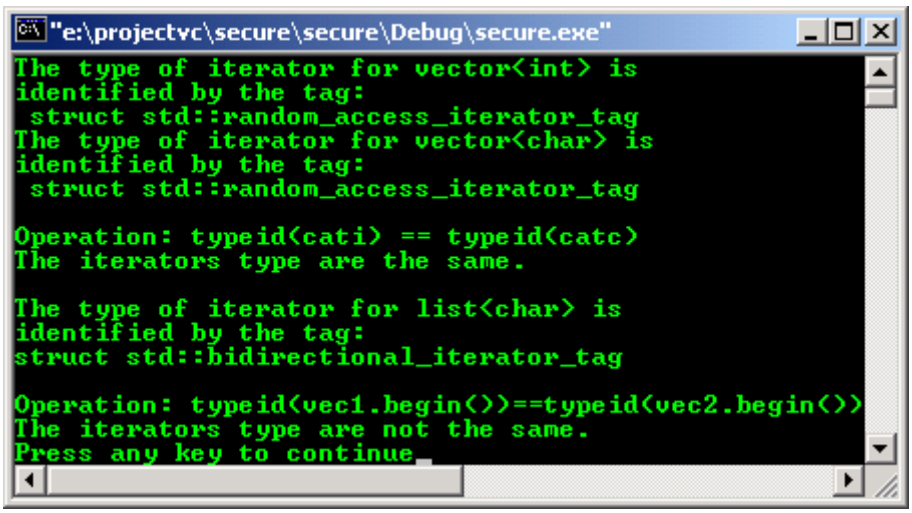
```
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

- The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time.
- For every iterator of type `Iterator`, `iterator_traits<Iterator>::iterator_category` must be defined to be the most specific category tag that describes the iterator's behavior.
- The type is the same as `iterator<Iter>::iterator_category` when `Iter` describes an object that can serve as a random-access iterator.

```
//iterator, template class
#include <iterator>
#include <vector>
#include <iostream>
#include <list>
using namespace std;

int main()
{
vector<int> vec1;
vector<char> vec2;
list<char> lst;
iterator_traits<vector<int>::iterator>::iterator_category cati;
iterator_traits<vector<char>::iterator>::iterator_category catc;
iterator_traits<list<char>::iterator>::iterator_category catlst;

//both are random-access iterators
cout<<"The type of iterator for vector<int> is\n"
<<"identified by the tag:\n"
<<" "<<typeid(cati).name()<<endl;

cout<<"The type of iterator for vector<char> is \n"
<<"identified by the tag:\n"
<<" "<<typeid(catc).name()<<"\n";

cout<<"\nOperation: typeid(cati) == typeid(catc)\n";
if(typeid(cati) == typeid(catc))
cout<<"The iterators type are the same."<<endl<<endl;
else
cout<<"The iterators type are not the same."<<endl<<endl;

//But the list iterator is bidirectional, not random access
cout<<"The type of iterator for list<char> is\n"
<<"identified by the tag:\n"
<<typeid(catlst).name()<<endl;

cout<<"\nOperation: typeid(vec1.begin())==typeid(vec2.begin())\n";
if(typeid(vec1.begin()) == typeid(vec2.begin()))
cout<<"The iterators type are the same."<<endl;
else
cout<<"The iterators type are not the same."<<endl;
```

```
return 0;
}
```

Output:



`iterator_traits` Template Class

- A template helper class providing critical types that are associated with different iterator types so that they can be referred to in the same way.
- Note that, from the following class template you can see how the `typedef` defined in the template class as well as `struct` usage.

```
template<class Iterator>
   struct iterator_traits
  {
   typedef typename Iterator::iterator_category iterator_category;
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::difference_type difference_type;
   typedef typename Iterator::pointer pointer;
   typedef typename Iterator::reference reference;
   };
template<class Type>
   struct iterator_traits<Type*>
  {
   typedef random_access_iterator_tag iterator_category;
   typedef Type value_type;
   typedef ptrdiff_t difference_type;
   typedef Type *pointer;
   typedef Type& reference;
   };
template<class Type>
   struct iterator_traits<const Type*>
  {
   typedef random_access_iterator_tag iterator_category;
   typedef Type value_type;
   typedef ptrdiff_t difference_type;
   typedef const Type *pointer;
   typedef const Type& reference;
   };
```

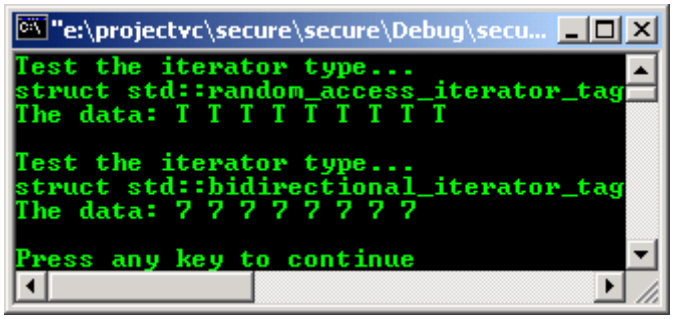
- The template class defines the member types:

Type	Description
`iterator_category`	A synonym for `Iterator::iterator_category`.
`value_type`	A synonym for `Iterator::value_type`.
`difference_type`	A synonym for `Iterator::difference_type`.
`pointer`	A synonym for `Iterator::pointer`.
`reference`	A synonym for `Iterator::reference`.

Table 31.17: Member types

```
//iterator, template class
#include <iostream>
#include <iterator>
#include <vector>
#include <list>
using namespace std;

template<class ite>
//create a function of template class type...
void funct(ite i1, ite i2)
{
iterator_traits<ite>::iterator_category cat;
cout<<"Test the iterator type...\n";
cout<<typeid(cat).name()<<endl;

//print the container data
cout<<"The data: ";
while(i1 != i2)
{
iterator_traits<ite>::value_type p;
p = *i1;
cout<<p<<" ";
i1++;
};
cout<<endl<<endl;
};

int main()
{
//declare containers vector and list
vector<char> vec(9, 'T');
list<int> lst(8, 7);

//function call...
funct(vec.begin(), vec.end());
funct(lst.begin(), lst.end());
return 0;
}
```

Output:



`insert_iterator` Template Class

- Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence and associative containers.
- The insert_iterator class is templatized on the type of container being adapted.

```
template <class Container>
```

Parameters

Parameter	Description
`Container`	The type of container into which elements are to be inserted by an `insert_iterator`.

Table 31.18

- The container of type `Container` must satisfy the requirements for a variable-sized container and have a two-argument insert member function where the parameters are of type

`Container::iterator` and `Container::value_type` and that returns a type `Container::iterator`.

- STL sequence and sorted associative containers satisfy these requirements and can be adapted to use with `insert_iterator`s. For associative containers, the position argument is treated as a hint, which has the potential to improve or degrade performance depending on how good the hint is.
- An `insert_iterator` must always be initialized with its container.

`insert_iterator` Template Class Members

Typedefs

Typedef	Description
`container_type`	A type that represents the container into which a general insertion is to be made.
`reference`	A type that provides a reference to an element in a sequence controlled by the associated container.

Table 31.19

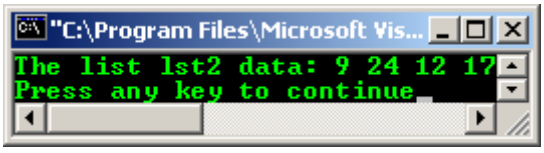
- A type that represents the container into which a general insertion is to be made.

```
typedef Container container_type;
```

- The type is a synonym for the template parameter `Container`.

```
//insert_iterator, container_type
#include <iterator>
#include <list>
#include <iostream>
using namespace std;

int main()
{
list<int> lst1;
insert_iterator<list<int> >::container_type lst2 = lst1;
inserter(lst2, lst2.end()) = 12;
inserter(lst2, lst2.end()) = 17;
inserter(lst2, lst2.begin()) = 24;
inserter(lst2, lst2.begin()) = 9;

list<int>::iterator veciter;
cout<<"The list lst2 data: ";
for(veciter = lst2.begin(); veciter != lst2.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;
return 0;
}
```

`Output:`



- A type that provides a reference to an element in a sequence controlled by the associated container.

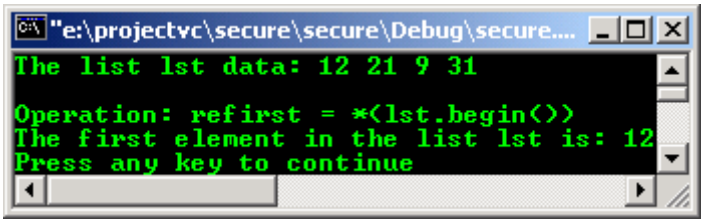
```
typedef typename Container::reference reference;
```

- The type describes a reference to an element of the sequence controlled by the associated container.

```
//insert_iterator, container_reference
#include <iterator>
#include <list>
#include <iostream>
using namespace std;

int main()
{
```

```
list<int> lst;
insert_iterator<list<int> > iivIter(lst, lst.begin());
*iivIter = 12;
*iivIter = 21;
*iivIter = 9;
*iivIter = 31;

list<int>::iterator lstIter;
cout<<"The list lst data: ";
for(lstIter = lst.begin(); lstIter != lst.end(); lstIter++)
cout<<*lstIter<<" ";
cout<<endl;

cout<<"\nOperation: refirst = *(lst.begin())\n";
insert_iterator<list<int> >::reference refirst = *(lst.begin());
cout<<"The first element in the list lst is: "<<refirst<<endl;
return 0;
}
```

Output:



Member Functions

Member function	Description
`insert_iterator`	Constructs an `insert_iterator` that inserts an element into a specified position in a container.

Table 31.20

`insert_iterator::insert_iterator`

- Constructs an `insert_iterator` that inserts an element into a specified position in a container.

```
insert_iterator(Container& _Cont, typename Container::iterator _It);
```

Parameters

Parameter	Description
`_Cont`	The container into which the `insert_iterator` is to insert elements.
`_It`	The position for the insertion.

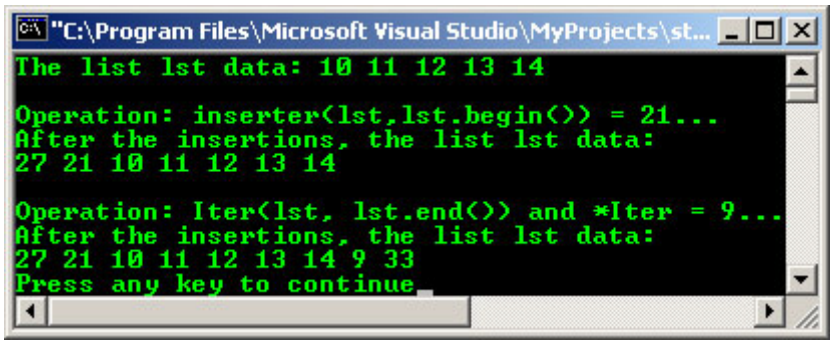
Table 31.21

- All containers have the insert member function called by the `insert_iterator`.
- For associative containers the position parameter is merely a suggestion. The inserter function provides a convenient way to insert to values.

```
//insert_iterator, insert_iterator
#include <iterator>
#include <list>
#include <iostream>

int main()
{
using namespace std;
int i;
list <int>::iterator lstiter;

list<int> lst;
for(i = 10; i<15; ++i)
lst.push_back(i);

cout<<"The list lst data: ";
```

```
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Using the member function to insert an element
cout<<"\nOperation: inserter(lst, lst.begin()) = 21...";
inserter(lst, lst.begin()) = 21;
inserter(lst, lst.begin()) = 27;

cout<<"\nAfter the insertions, the list lst data:\n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Alternatively, using the template version
cout<<"\nOperation: Iter(lst, lst.end()) and *Iter = 9...";
insert_iterator< list < int> > Iter(lst, lst.end());
*Iter = 9;
*Iter = 33;

cout<<"\nAfter the insertions, the list lst data:\n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;
return 0;
}
```

Output:



Operators

Operator	Description
`operator*`	Dereferencing operator used to implement the output iterator expression such as `*i = x` for a general insertion.
`operator++`	Increments the `insert_iterator` to the next location into which a value may be stored.
`operator=`	Assignment operator used to implement the output iterator expression such as `*i = x` for a general insertion.

Table 31.22

`insert_iterator::operator*`

- Dereferences the insert iterator returning the element is addresses.

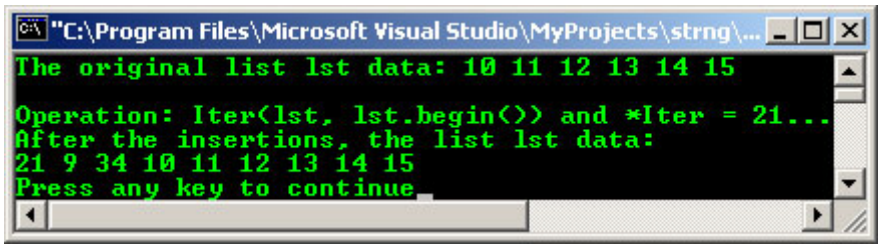
```
insert_iterator& operator*();
```

- The return value is the member function returns the value of the element addressed.
- Used to implement the output iterator expression `*Iter = value`. If `Iter` is an iterator that addresses an element in a sequence, then `*Iter = value` replaces that element with value and does not change the total number of elements in the sequence.

```
//insert_iterator, operator*
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
int i;
list<int>::iterator lstiter;

list<int> lst;
for(i = 10; i<=15; ++i)
lst.push_back(i);

cout<<"The original list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

cout<<"\nOperation: Iter(lst, lst.begin()) and *Iter = 21... \n";
insert_iterator< list < int> > Iter(lst, lst.begin());
*Iter = 21;
*Iter = 9;
*Iter = 34;

cout << "After the insertions, the list lst data:\n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;
return 0;
}
```

Output:



- Program example compiled using g++. g++ will prompt you if old STL constructs that do not comply to standard, used in your programs such as examples presented at the beginning of this Module.

```
//*****iteratoradvance.cpp**********
//iterator, advance()
#include <iterator>
#include <list>
#include <iostream>
using namespace std;

int main()
{
int i;

list<int> lst;
for(i = 1; i <= 10; ++i)
lst.push_back(i);

list<int>::iterator lstIter, lstpos = lst.begin();

cout<<"The lst list data: ";
for(lstIter = lst.begin(); lstIter != lst.end(); lstIter++)
cout<<*lstIter<<" ";
cout<<endl;
cout<<"The first element pointed by iterator lstpos is: "<<*lstpos<<endl;

advance(lstpos, 5);
cout<<"Advanced lstpos 5 steps forward pointing to the "<<*lstpos<<endl;

advance(lstpos, -4);
cout<<"Moved lstpos 4 steps backward pointing to the "<<*lstpos<<endl;

advance(lstpos, 8);
cout<<"Finally, the last element pointed by iterator lstpos is: "<<*lstpos<<endl;

return 0;
}
```

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

```
The lst list data: 1 2 3 4 5 6 7 8 9 10
The first element pointed by iterator lstpos is: 1
Advanced lstpos 5 steps forward pointing to the 6
Moved lstpos 4 steps backward pointing to the 2
Finally, the last element pointed by iterator lstpos is: 10
```

```
//*******insertiterator.cpp********
//insert_iterator, insert_iterator
#include <iterator>
#include <list>
#include <iostream>

int main()
{
using namespace std;
int i;
list <int>::iterator lstiter;

list<int> lst;
for(i = 10; i<15; ++i)
lst.push_back(i);

cout<<"The list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Using the member function to insert an element
cout<<"\nOperation: inserter(lst, lst.begin()) = 21...";
inserter(lst, lst.begin()) = 21;
inserter(lst, lst.begin()) = 27;

cout<<"\nAfter the insertions, the list lst data:\n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

//Alternatively, using the template version
cout<<"\nOperation: Iter(lst, lst.end()) and *Iter = 9...";
insert_iterator< list < int> > Iter(lst, lst.end());
*Iter = 9;
*Iter = 33;

cout<<"\nAfter the insertions, the list lst data:\n";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;
return 0;
}
```

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

```
The list lst data: 10 11 12 13 14

Operation: inserter(lst, lst.begin()) = 21...
After the insertions, the list lst data:
27 21 10 11 12 13 14

Operation: Iter(lst, lst.end()) and *Iter = 9...
After the insertions, the list lst data:
27 21 10 11 12 13 14 9 33
```

- *To be continued on next Module...*

Further reading and digging:

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