MODULE 32
--THE STL--
ITERATOR PART II

My Training Period: hours

Note: Compiled using VC++7.0/.Net, win32 empty console mode application. This is a continuation from the previous Module. **g++** compilation examples given at the end of this Module.

Abilities

- Able to understand and use iterator template classes.
- Able to understand and use iterator adapters.
- Able to understand and use stream iterator.

1.1 Continuation from the previous Module

insert_iterator::operator++

- Increments the `insert_iterator` to the next location into which a value may be stored.

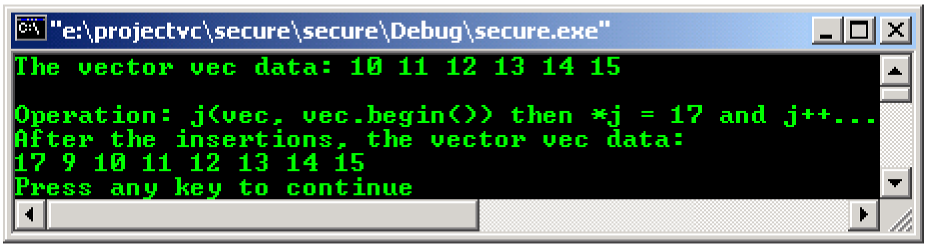
```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

Parameters

- An `insert_iterator` addressing the next location into which a value may be stored.
- Both pre-incrementation and post-incrementation operators return the same result.

```
//insert_iterator, operator++
//the increment...
#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: ";
//iterate all the elements and print...
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

cout<<"\nOperation: j(vec, vec.begin()) then *j = 17 and j++...\n";
insert_iterator<vector<int> > j(vec, vec.begin());
*j = 17;
j++;
*j = 9;

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

Output:



```
The vector vec data: 10 11 12 13 14 15

Operation: j(vec, vec.begin()) then *j = 17 and j++...
After the insertions, the vector vec data:
17 9 10 11 12 13 14 15
Press any key to continue
```

insert_iterator::operator=

- Assignment operator used to implement the output iterator expression such as `*i = x`.

```
insert_iterator& operator=(typename Container::const_reference _Val);
```

Parameter

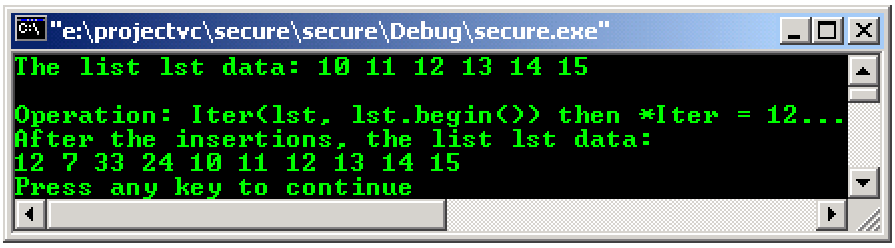
Parameter	Description
`_Val`	The value to be assigned to the element.

Table 32.1

- The return value is a reference to the element inserted into the container.
- The member function evaluates `Iter = container. insert(Iter, _Val)`, then returns `*this`.

```
//insert_iterator, operator=
//the assignment
#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 list lst data: ";
for(lstiter = lst.begin(); lstiter != lst.end(); lstiter++)
cout<<*lstiter<<" ";
cout<<endl;

insert_iterator< list < int> > Iter(lst, lst.begin());
*Iter = 12;
*Iter = 7;
*Iter = 33;
*Iter = 24;

cout<<"\nOperation: Iter(lst, lst.begin()) then *Iter = 12...\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:



```
The list lst data: 10 11 12 13 14 15

Operation: Iter(lst, lst.begin()) then *Iter = 12...
After the insertions, the list lst data:
12 7 33 24 10 11 12 13 14 15
Press any key to continue
```

istream_iterator Template Class

- Describes an input iterator object. It extracts objects of class Type from an input stream, which it accesses through an object it stores, of type pointer to basic_istream<CharType, Traits>.

```
template <
   class Type
   class CharType = char
   class Traits = char_traits<CharType>
   class Distance= ptrdiff_t
>
```

Parameters

Parameter	Description
Type	The type of object to be extracted from the input stream.
CharType	The type that represents the character type for the istream_iterator. This argument is optional and the default value is char.
Traits	The type that represents the character type for the istream_iterator. This argument is optional and the default value is char_traits<CharType>.
Distance	A signed integral type that represents the difference type for the istream_iterator. This argument is optional and the default value is ptrdiff_t.

Table 32.2

- After constructing or incrementing an object of class istream_iterator with a non null stored pointer, the object attempts to extract and store an object of type Type from the associated input stream.
- If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

istream_iterator Template Class Members

Typedefs

Typedef	Description
char_type	A type that provides for the character type of the istream_iterator.
istream_type	A type that provides for the stream type of the istream_iterator.
traits_type	A type that provides for the character traits type of the istream_iterator.

Table 32.3

istream_iterator::char_type

- A type that provides for the character type of the istream_iterator.

```
typedef CharType char_type;
```

- The type is a synonym for the template parameter CharType.

istream_iterator::traits_type

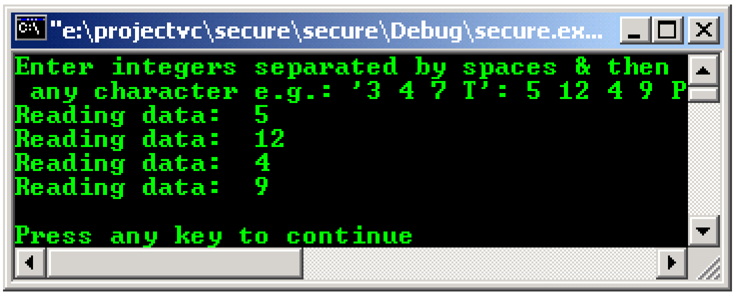
- A type that provides for the character traits type of the istream_iterator.

```
typedef Traits traits_type;
```

- The type is a synonym for the template parameter Traits.

```
//istream_iterator, char_type and
//traits_type
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
typedef istream_iterator<int>::char_type chtype;
typedef istream_iterator<int>::traits_type tratype;

//Standard iterator interface for reading
//elements from the input stream...
cout<<"Enter integers separated by spaces & then\n"
<<" any character e.g.: '3 4 7 T': ";

//istream_iterator for reading int stream
istream_iterator<int, chtype, tratype> intread(cin);

//End-of-stream iterator
istream_iterator<int, chtype, tratype> EOFintread;

while(intread != EOFintread)
{
cout<<"Reading data:  "<<*intread<<endl;
++intread;
}
cout<<endl;
return 0;
}
```

Output:



```
Enter integers separated by spaces & then
 any character e.g.: '3 4 7 T': 5 12 4 9 P
Reading data:  5
Reading data:  12
Reading data:  4
Reading data:  9

Press any key to continue
```

Member Functions

Member function	Description
`istream_iterator`	Constructs either an end-of-stream iterator as the default `istream_iterator` or a `istream_iterator` initialized to the iterator's stream type from which it reads.

Table 32.4

istream_iterator::istream_iterator

- Constructs either an end-of-stream iterator as the default `istream_iterator` or a `istream_iterator` initialized to the iterator's stream type from which it reads.

```
istream_iterator();
istream_iterator(istream_type& _Istr);
```

Parameter

Parameter	Description
`_Istr`	The input stream to be read use to initialize the `istream_iterator`.

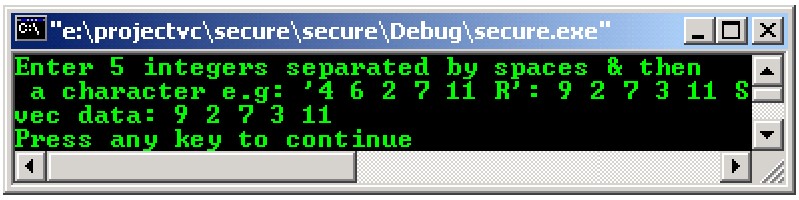
Table 32.5

- The First constructor initializes the input stream pointer with a null pointer and creates an end-of-stream iterator.

- The second constructor initializes the input stream pointer with &_Istr, then attempts to extract and store an object of type Type.
- The end-of-stream iterator can be use to test whether an istream_iterator has reached the end of a stream.

```
//istream_iterator, istream_iterator
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
//Used in conjunction with copy algorithm
//to put elements into a vector read from cin
vector<int> vec(5);
vector<int>::iterator Iter;

cout<<"Enter 5 integers separated by spaces & then\n"
<<" a character e.g: '4 6 2 7 11 R': ";
istream_iterator<int> intvecread(cin);

//Default constructor will test equal to end of stream
//for delimiting source range of vector
copy(intvecread, istream_iterator<int>(), vec.begin());
cin.clear();

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

Output:



```
Enter 5 integers separated by spaces & then
 a character e.g: '4 6 2 7 11 R': 9 2 7 3 11 S
vec data: 9 2 7 3 11
Press any key to continue
```

Operators

Operator	Description
operator*	The dereferencing operator returns the stored object of type Type addressed by the istream_iterator.
operator->	Returns the value of a member, if any.
operator++	Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

Table 32.6

istreambuf_iterator Template Class

- The template class istreambuf_iterator describes an input iterator object that extracts character elements from an input stream buffer, which it accesses through an object it stores, of type pointer to basic_streambuf<CharType, Traits>.

```
template <
   class CharType
   class Traits = char_traits<CharType>
>
```

Parameters

Parameter	Description
CharType	The type that represents the character type for the

	istreambuf_iterator.
Traits	The type that represents the character type for the istreambuf_iterator. This argument is optional and the default value is char_traits<CharType>.

Table 32.7

- The ostreambuf_iterator class must satisfy the requirements for an input iterator.
- After constructing or incrementing an object of class istreambuf_iterator with a non-null stored pointer, the object effectively attempts to extract and store an object of type CharType from the associated input stream.
- The extraction may be delayed, however, until the object is actually dereferenced or copied. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

istreambuf_iterator Template Class Members

Typedefs

Typedef	Description
char_type	A type that provides for the character type of the ostreambuf_iterator.
int_type	A type that provides an integer type for an istreambuf_iterator.
istream_type	A type that provides for the stream type of the istream_iterator.
streambuf_type	A type that provides for the stream type of the istreambuf_iterator.
traits_type	A type that provides for the character traits type of the istream_iterator.

Table 32.8

istreambuf_iterator::char_type

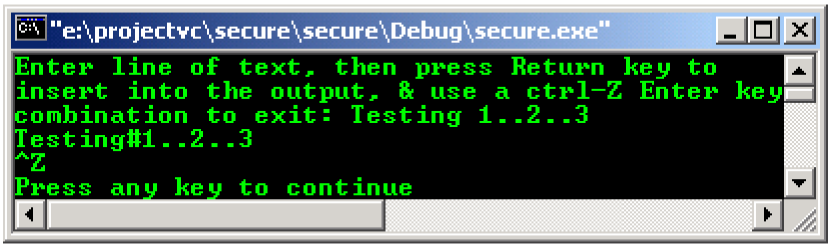
- A type that provides for the character type of the ostreambuf_iterator.

```
typedef CharType char_type;
```

- The type is a synonym for the template parameter CharType.

```
//istreambuf_iterator, char_type
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
typedef istreambuf_iterator<char>::char_type chatype;
typedef istreambuf_iterator<char>::traits_type tratype;

cout<<"Enter line of text, then press Return key to \n"
<<"insert into the output, & use a ctrl-Z Enter key\n"
<<"combination to exit: ";

//istreambuf_iterator for input stream
istreambuf_iterator< chatype, tratype> charInBuf(cin);
ostreambuf_iterator<char> charOut(cout);

//Used in conjunction with replace_copy algorithm
//to insert into output stream and replaces spaces
//with hash sign
replace_copy(charInBuf, istreambuf_iterator<char>(), charOut, ' ', '#');
return 0;
}
```

Output:



```
Enter line of text, then press Return key to
insert into the output, & use a ctrl-Z Enter key
combination to exit: Testing 1..2..3
Testing#1..2..3
^Z
Press any key to continue
```

istreambuf_iterator::int_type

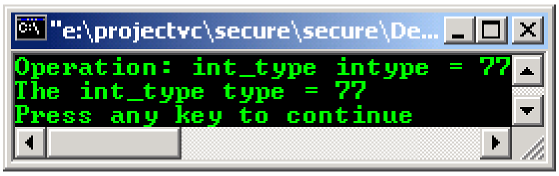
- A type that provides an integer type for an `istreambuf_iterator`.

```
typedef typename Traits::int_type int_type;
```

- The type is a synonym for `Traits::int_type`.

```
//istreambuf_iterator, int_type
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
cout<<"Operation: int_type intype = 77\n";
istreambuf_iterator<char>::int_type intype = 77;
cout<<"The int_type type = "<<intype<<endl;
return 0;
}
```

Output:



```
Operation: int_type intype = 77
The int_type type = 77
Press any key to continue
```

istream_iterator::traits_type

- A type that provides for the character traits type of the `istream_iterator`.

```
typedef Traits traits_type;
```

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

Member Functions

Member function	Description
`equal()`	Tests for equality between two input stream buffer iterators.
`istreambuf_iterator`	Constructs an `istreambuf_iterator` that is initialized to read characters from the input stream.

Table 32.9

istreambuf_iterator::equal

- Tests for equivalence between two input stream buffer iterators.

```
bool equal(const istreambuf_iterator& _Right) const;
```

Parameter

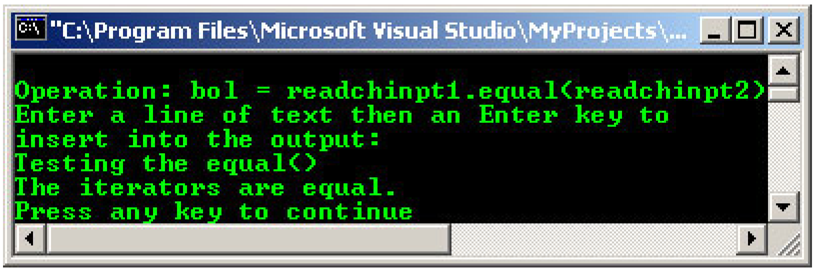
Parameter	Description
`_Right`	The iterator for which to check for equality.

Table 32.10

- The return value is **true** if both `istreambuf_iterator`s are end-of-stream iterators or if neither is an end-of-stream iterator; otherwise **false**.
- A range is defined by the `istreambuf_iterator` to the current position and the end-of-stream iterator, but since all non-end-of stream iterators are equivalent under the `equal()` member function, it is not possible to define any sub-ranges using `istreambuf_iterators`.
- The == and != operators have the same semantics.

```
//istreambuf_iterator, equal
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
cout<<"\nOperation: bol = readchinpt1.equal(readchinpt2)\n";
cout<<"Enter a line of text then an Enter key to\n"
<<"insert into the output:\n";

istreambuf_iterator<char> readchinpt1(cin);
istreambuf_iterator<char> readchinpt2(cin);

bool bol = readchinpt1.equal(readchinpt2);
if(bol)
cout<<"The iterators are equal."<<endl;
else
cout<<"The iterators are not equal."<<endl;
return 0;
}
```

Output:



```
Operation: bol = readchinpt1.equal(readchinpt2)
Enter a line of text then an Enter key to
insert into the output:
Testing the equal()
The iterators are equal.
Press any key to continue
```

istreambuf_iterator::istreambuf_iterator

- Constructs an `istreambuf_iterator` that is initialized to read characters from the input stream.

```
istreambuf_iterator
(
   streambuf_type* _Strbuf = 0
) throw();
istreambuf_iterator
(
   istream_type& _Istr
) throw();
```

Parameters

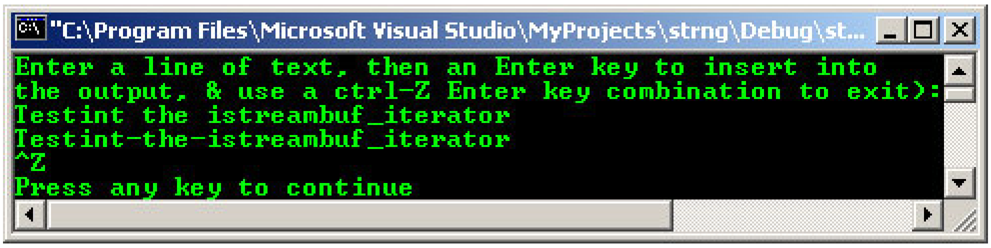
Parameter	Description
`_Strbuf`	The input stream buffer to which the `istreambuf_iterator` is being attached.
`_Istr`	The input stream to which the `istreambuf_iterator` is being attached.

Table 32.11

- The first constructor initializes the input stream-buffer pointer with `_Strbuf`.
- The second constructor initializes the input stream-buffer pointer with `_Istr.rdbuf`, and then eventually attempts to extract and store an object of type `CharType`.

```
//istreambuf_iterator, istreambuf_iterator
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
istreambuf_iterator<char>::istream_type &istrm = cin;
istreambuf_iterator<char>::streambuf_type *strmbf = cin.rdbuf();

cout<<"Enter a line of text, then an Enter key to insert into\n"
<<"the output, (& use a ctrl-Z Enter key combination to exit):\n";

istreambuf_iterator<char> charReadIn(cin);
ostreambuf_iterator<char> charOut(cout);

//Used in conjunction with replace_copy algorithm
//to insert into output stream and replace spaces
//with hyphen-separators
replace_copy(charReadIn, istreambuf_iterator<char>(), charOut, ' ', '-');
return 0;
}
```

Output:


"C:\Program Files\Microsoft Visual Studio\MyProjects\strng\Debug\st...

```
Enter a line of text, then an Enter key to insert into
the output, & use a ctrl-Z Enter key combination to exit):
Testint the istreambuf_iterator
Testint-the-istreambuf_iterator
^Z
Press any key to continue
```

Operators

Operator	Description
`operator*`	The dereferencing operator returns the next character in the stream.
`operator++`	Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.
`operator->`	Returns the value of a member, if any.

Table 32.12

istreambuf_iterator::operator++

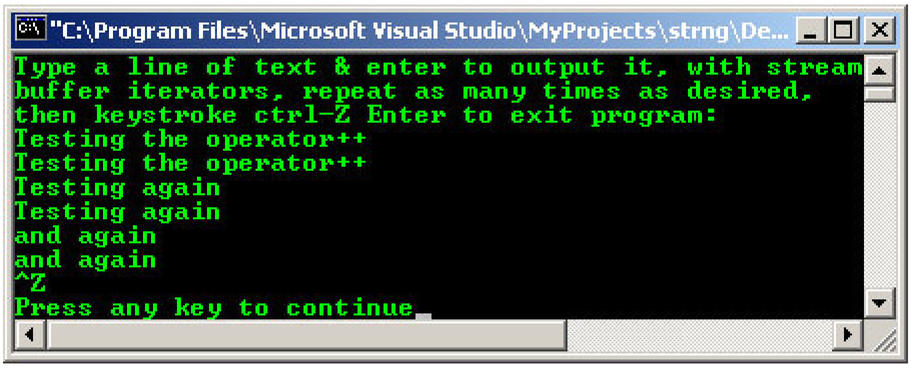
- Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.

```
istreambuf_iterator& operator++();
istreambuf_iterator operator++(int);
```

- The return value is an `istreambut_iterator` or a reference to an `istreambuf_iterator`.
- The first operator eventually attempts to extract and store an object of type `CharType` from the associated input stream.
- The second operator makes a copy of the object, increments the object, and then returns the copy.

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

int main()
{
cout<<"Type a line of text & enter to output it, with stream\n"
<<"buffer iterators, repeat as many times as desired,\n"
<<"then keystroke ctrl-Z Enter to exit program: \n";

istreambuf_iterator<char> inpos(cin);
istreambuf_iterator<char> endpos;
ostreambuf_iterator<char> outpos(cout);

while(inpos != endpos)
{
*outpos = *inpos;
//Increment istreambuf_iterator
++inpos;
++outpos;
}
   return 0;
}
```

Output:



```
Type a line of text & enter to output it, with stream
buffer iterators, repeat as many times as desired,
then keystroke ctrl-Z Enter to exit program:
Testing the operator++
Testing the operator++
Testing again
Testing again
and again
and again
^Z
Press any key to continue
```

"C:\Program Files\Microsoft Visual Studio\MyProjects\strng\De...

ostream_iterator Template Class

- The template class ostream_iterator describes an output iterator object that writes successive elements onto the output stream with the extraction operator >>.

```
template <
   class Type
   class CharType = char
   class Traits = char_traits<CharType>
>
```

Parameters

Parameter	Description
Type	The type of object to be inserted into the output stream.
CharType	The type that represents the character type for the ostream_iterator. This argument is optional and the default value is char.
Traits	The type that represents the character type for the ostream_iterator. This argument is optional and the default value is char_traits<CharType>.

Table 32.13

- The ostream_iterator class must satisfy the requirements for an output iterator.
- Algorithms can be written directly to output streams using an ostream_iterator.

ostream_iterator Template Class Members

Typedefs

Typedef	Description
char_type	A type that provides for the character type of the ostream_iterator.
ostream_type	A type that provides for the stream type of the ostream_iterator.
traits_type	A type that provides for the character traits type of the ostream_iterator.

Table 32.14

ostream_iterator::ostream_iterator

- Constructs an `ostream_iterator` that is initialized and delimited to write to the output stream.

```
ostream_iterator(ostream_type& _Ostr);
ostream_iterator(ostream_type& _Ostr, const CharType* _Delimiter);
```

Parameters

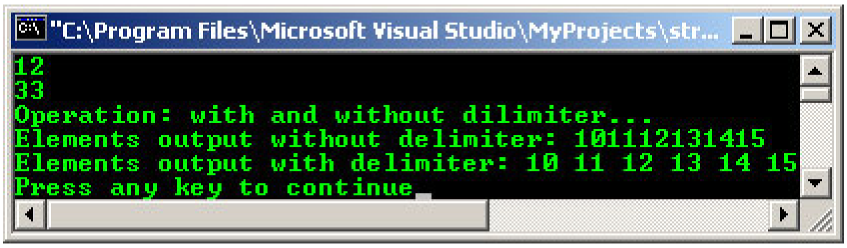
Parameter	Description
`_Ostr`	The output stream object used to initialize the output stream pointer.
`_Delimiter`	The output stream delimiter used to initialize the output stream pointer.

Table 32.15

- The first constructor initializes the output stream pointer with `&_Ostr`. The delimiter string pointer designates an empty string.
- The second constructor initializes the output stream pointer with `&_Ostr` and the delimiter string pointer with `_Delimiter`.

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

int main()
{
//ostream_iterator for stream cout
ostream_iterator<int> intOut(cout, "\n");
*intOut = 12;
intOut++;
*intOut = 33;
intOut++;

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

cout<<"Operation: with and without delimiter...\n";
//Write elements to standard output stream
cout<<"Elements output without delimiter: ";
copy(vec.begin(), vec.end(), ostream_iterator<int> (cout));
cout<<endl;

//Write elements with delimiter " " to output stream
cout<<"Elements output with delimiter: ";
copy(vec.begin(), vec.end(), ostream_iterator<int> (cout, " "));
cout<<endl;
return 0;
}
```

Output:



```
12
33
Operation: with and without dilimiter...
Elements output without delimiter: 101112131415
Elements output with delimiter: 10 11 12 13 14 15
Press any key to continue
```

Member Functions

Member function	Description
`ostream_iterator`	Constructs an `ostream_iterator` that is initialized and delimited to write to the output stream.

Table 32.16

Operators

Operator	Description
`operator*`	Dereferencing operator used to implement the output iterator expression such as *i = x.
`operator++`	A nonfunctional increment operator that returns an `ostream_iterator` to the same object it addressed before the operation was called.
`operator=`	Assignment operator used to implement the output iterator expression such as `*i = x` for writing to an output stream.

Table 32.17

ostream_iterator::operator=

- Assignment operator used to implement the `output_iterator` expression such as `*i = x` for writing to an output stream.

```
ostream_iterator<Type, CharType, Traits>& operator=(const Type& _Val);
```

Parameter

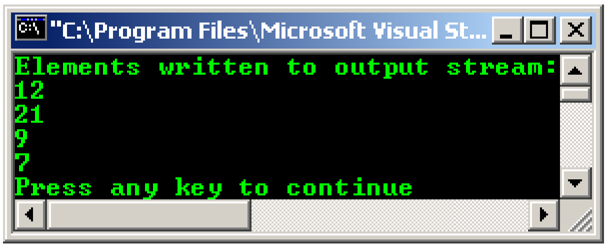
Parameter	Description
`_Val`	The value of the object of type `Type` to be inserted into the output stream.

Table 32.18

- The return value is the operator inserts `_Val` into the output stream associated with the object, and then returns a reference to the `ostream_iterator`.
- The requirements for an output iterator that the `ostream_iterator` must satisfy require only the expression such as`*j = t` be valid and says nothing about the operator or the operator= on their own.
- This member operator returns `*this`.

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

int main()
{
//ostream_iterator for stream cout
//with new line delimiter
ostream_iterator<int> intOut(cout, "\n");

//Standard iterator interface for writing
//elements to the output stream
cout<<"Elements written to output stream:\n";
*intOut = 12;
*intOut = 21;
//No effect on iterator position
intOut++;
*intOut = 9;
*intOut = 7;
return 0;
}
```

Output:



```
Elements written to output stream:
12
21
9
7
Press any key to continue
```

`ostreambuf_iterator` Template Class

- The template class `ostreambuf_iterator` describes an output iterator object that writes successive character elements onto the output stream with the extraction `operator>>`.
- The `ostreambuf_iterators` differ from those of the `ostream_iterator` Class in having characters instead of a generic type at the type of object being inserted into the output stream.

```
template <
   class CharType = char
   class Traits = char_traits<CharType>
>
```

Parameters

Parameter	Description
`CharType`	The type that represents the character type for the `ostreambuf_iterator`. This argument is optional and the default value is `char`.
`Traits`	The type that represents the character type for the `ostreambuf_iterator`. This argument is optional and the default value is `char_traits<CharType>`.

Table 32.19

- The `ostreambuf_iterator` class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an `ostreambuf_iterator`.
- The class provides a low-level stream iterator that allows access to the raw (unformatted) I/O stream in the form of characters and the ability to bypass the buffering and character translations associated with the high-level stream iterators.

`ostreambuf_iterator` Template Class Members

Typedefs

Typedef	Description
`char_type`	A type that provides for the character type of the `ostreambuf_iterator`.
`ostream_type`	A type that provides for the stream type of the `ostream_iterator`.
`streambuf_type`	A type that provides for the stream type of the `ostreambuf_iterator`.
`traits_type`	A type that provides for the character traits type of the `ostream_iterator`.

Table 32.20

`ostreambuf_iterator::ostreambuf_iterator`

- Constructs an `ostreambuf_iterator` that is initialized to write characters to the output stream.

```
ostreambuf_iterator(streambuf_type* _Strbuf) throw();
ostreambuf_iterator(ostream_type& _Ostr) throw();
```

Parameters

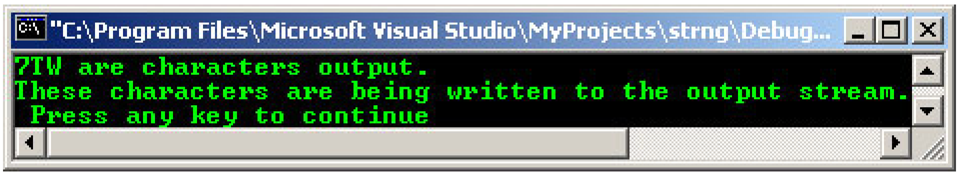
Parameter	Description
`_Strbuf`	The output `streambuf` object used to initialize the output stream-buffer pointer.
`_Ostr`	The output stream object used to initialize the output stream-buffer pointer.

Table 32.21

- The first constructor initializes the output stream-buffer pointer with `_Strbuf`.
- The second constructor initializes the output stream-buffer pointer with `_Ostr.rdbuf`.
- The stored pointer must not be a null pointer.

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

int main()
{
// ostreambuf_iterator for stream cout
ostreambuf_iterator<char> charOut(cout);

*charOut = '7';
charOut ++;
*charOut  = 'T';
charOut ++;
*charOut = 'W';
cout<<" are characters output."<<endl;

ostreambuf_iterator<char> strOut(cout);
string str = "These characters are being written to the output stream.\n ";
copy(str.begin(), str.end(), strOut);
return 0;
}
```

Output:


"C:\Program Files\Microsoft Visual Studio\MyProjects\strng\Debug...

```
7TW are characters output.
These characters are being written to the output stream.
 Press any key to continue
```

Member Functions

Member function	Description
`failed()`	Tests for failure of an insertion into the output stream buffer.
`ostreambuf_iterator`	Constructs an `ostreambuf_iterator` that is initialized to write characters to the output stream.

Table 32.22

ostreambuf_iterator::failed

- Tests for failure of an insertion into the output stream buffer.

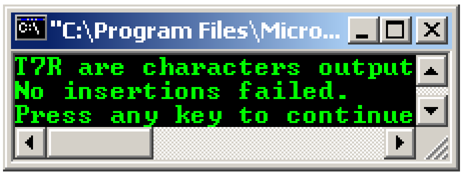
```
bool failed() const throw();
```

- The return value is **true** if no insertion into the output stream buffer has failed earlier; otherwise **false**.
- The member function returns **true** if, in any prior use of member `operator=`, the call to `subf_->sputc` returned `eof`.

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

int main()
{
```

```
//ostreambuf_iterator for stream cout
ostreambuf_iterator<char> charOut(cout);

*charOut = 'T';
charOut ++;
*charOut  = '7';
charOut ++;
*charOut = 'R';
cout<<" are characters output"<<endl;

bool bol = charOut.failed();
if(bol)
cout<<"At least one insertion failed."<<endl;
else
cout<<"No insertions failed."<<endl;
return 0;
}
```

Output:



```
T7R are characters output
No insertions failed.
Press any key to continue
```

Operators

Operator	Description
`operator*`	Dereferencing operator used to implement the output iterator expression such as `*i = x`.
`operator++`	A nonfunctional increment operator that returns an `ostreambuf_iterator` to the same object it addressed before the operation was called.
`operator=`	The operator inserts a character into the associated stream buffer.

Table 32.23

`reverse_iterator` Template Class

- The template class is an iterator adaptor that describes a reverse iterator object that behaves like a random-access or bidirectional iterator, only in reverse. It enables the backward traversal of a range.

```
template<class Iterator>
```

Parameter

Parameter	Description
`Iterator`	The type that represents the iterator to be adapted to operate in reverse.

Table 32.24

- Existing STL containers also define `reverse_iterator` and `const_reverse_iterator` types and have member functions `rbegin()` and `rend()` that return reverse iterators.
- These iterators have overwritten semantics. The `reverse_iterator` adaptor supplements this functionality as offers insert semantics and can also be used with streams.
- The `reverse_iterators` that require a bidirectional iterator must not call any of the member functions `operator+=`, `operator+`, `operator-=`, `operator-`, or `operator[]`, which may only be used with random-access iterators.
- If the range of an iterator is `[_First, _Last)`, where the square bracket on the left indicates the inclusion on `_First` and the parenthesis on the right indicates the inclusion of elements up to `_Left` but excluding `_Left` itself.
- The same elements are included in the reversed sequence `[rev – _First, rev – _Left)` so that if `_Left` is the one-past-the-end element in a sequence, then the first element `rev – _First` in the reversed sequence points to `*(_Left – 1)`. The identity which relates all reverse iterators to their underlying iterators is:

```
&*(reverse_iterator (i)) == &*(i – 1)
```

- In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence.
- So if an iterator addressed the element valued 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element valued 4 in the reversed sequence (8, 6, 4, 2).

`reverse_iterator` Template Class Members

Typedefs

Typedef	Description
`difference_type`	A type that provides the difference between two `reverse_iterators` referring to elements within the same container.
`iterator_type`	A type that provides the underlying iterator for a `reverse_iterator`.
`pointer`	A type that provides a pointer to an element addressed by a `reverse_iterator`.
`reference`	A type that provides a reference to an element addressed by a `reverse_iterator`.

Table 32.25

`reverse_iterator::operator[]`

- Returns a reference to an element offset from the element addressed by a `reverse_iterator` by a specified number of positions.

```
reference operator[](difference_type _Off) const;
```

Parameter

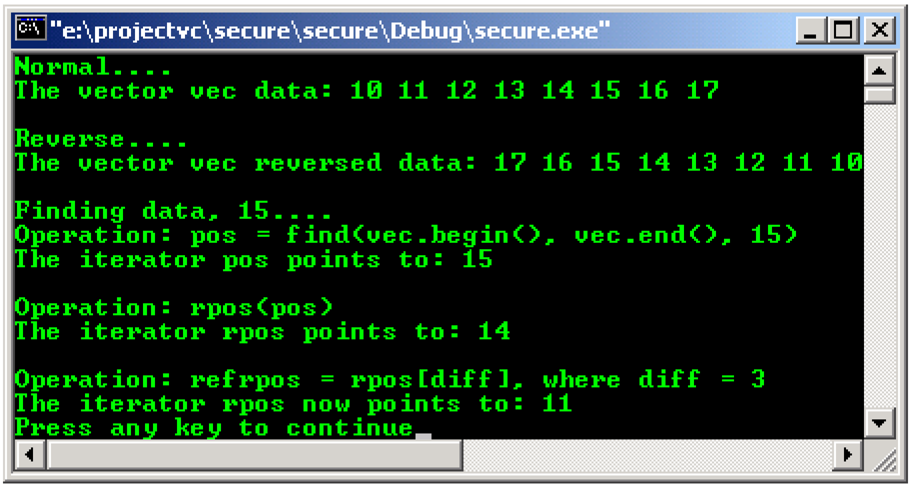
Parameter	Description
`_Off`	The offset from the reverse_iterator address.

Table 32.26

- The return value is the reference to the element offset.
- The operator returns `*(*this + _Off)`.

```
//reverse_iterator, operator[]
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
int i;

vector<int> vec;
for(i = 10; i<=17; ++i)
vec.push_back(i);

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

cout<<"\nReverse....\n";
vector <int>::reverse_iterator rvIter;
cout<<"The vector vec reversed data: ";
for(rvIter = vec.rbegin(); rvIter != vec.rend(); rvIter++)
cout<<*rvIter<<" ";
cout<<endl;

cout<<"\nFinding data, 15....\n";
cout<<"Operation: pos = find(vec.begin(), vec.end(), 15)\n";
vector <int>::iterator pos;
pos = find(vec.begin(), vec.end(), 15);

cout<<"The iterator pos points to: "<<*pos<<endl;
reverse_iterator<vector<int>::iterator> rpos(pos);

//Declare a difference type for a parameter
reverse_iterator<vector<int>::iterator>::difference_type diff = 3;

cout<<"\nOperation: rpos(pos)\n";
cout<<"The iterator rpos points to: "<<*rpos<<endl;

//Declare a reference return type & use operator[]
cout<<"\nOperation: refrpos = rpos[diff], where diff = 3\n";
reverse_iterator<vector<int>::iterator>::reference refrpos = rpos[diff];
cout<<"The iterator rpos now points to: "<<refrpos<<endl;
return 0;
}
```

Output:



```
Normal....
The vector vec data: 10 11 12 13 14 15 16 17

Reverse....
The vector vec reversed data: 17 16 15 14 13 12 11 10

Finding data, 15....
Operation: pos = find(vec.begin(), vec.end(), 15)
The iterator pos points to: 15

Operation: rpos(pos)
The iterator rpos points to: 14

Operation: refrpos = rpos[diff], where diff = 3
The iterator rpos now points to: 11
Press any key to continue
```

reverse_iterator::pointer

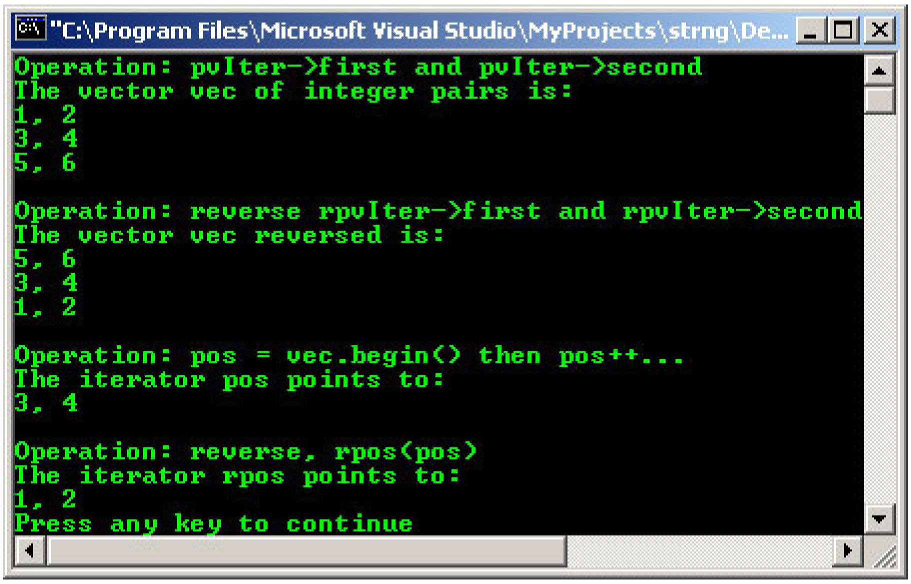
- A type that provides a pointer to an element addressed by a `reverse_iterator`.

```
typedef typename iterator_traits<Iterator>::pointer pointer;
```

- The type is a synonym for the iterator trait typename `iterator_traits<Random-access iterator>::pointer`.

```
//reverse_iterator, pointer
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>
using namespace std;

int main()
{

typedef vector<pair<int, int> > pVector;
pVector vec;
vec.push_back(pVector::value_type(1, 2));
vec.push_back(pVector::value_type(3, 4));
vec.push_back(pVector::value_type(5, 6));

pVector::iterator pvIter;
cout<<"Operation: pvIter->first and pvIter->second\n";
cout<<"The vector vec of integer pairs is: \n";
for(pvIter = vec.begin(); pvIter != vec.end(); pvIter++)
cout<<pvIter->first<<", "<<pvIter->second<<endl;

pVector::reverse_iterator rpvIter;
cout<<"\nOperation: reverse rpvIter->first and rpvIter->second";
cout<<"\nThe vector vec reversed is: \n";
for(rpvIter = vec.rbegin(); rpvIter != vec.rend(); rpvIter++)
cout<<rpvIter->first<< ", " <<rpvIter->second<<endl;

cout<<"\nOperation: pos = vec.begin() then pos++...";
pVector::iterator pos = vec.begin();
pos++;
cout<<"\nThe iterator pos points to:\n"<<pos->first<< ", " <<pos->second<<endl;

cout<<"\nOperation: reverse, rpos(pos)";
pVector::reverse_iterator rpos(pos);
cout<<"\nThe iterator rpos points to:\n"<<rpos->first<< ", " <<rpos->second<<endl;
return 0;
}
```

Output:



"C:\Program Files\Microsoft Visual Studio\MyProjects\strng\De...

```
Operation: pvIter->first and pvIter->second
The vector vec of integer pairs is:
1, 2
3, 4
5, 6

Operation: reverse rpvIter->first and rpvIter->second
The vector vec reversed is:
5, 6
3, 4
1, 2

Operation: pos = vec.begin() then pos++...
The iterator pos points to:
3, 4

Operation: reverse, rpos(pos)
The iterator rpos points to:
1, 2
Press any key to continue
```

Member Functions

Member function	Description
`base()`	Recovers the underlying iterator from its `reverse_iterator`.
`reverse_iterator`	Constructs a default `reverse_iterator` or a `reverse_iterator` from an underlying iterator.

Table 32.27

reverse_iterator::base

- Recovers the underlying iterator from its `reverse_iterator`.

```
RandomIterator base() const;
```

- The return value is the iterator underlying the `reverse_iterator`.
- The identity that relates all reverse iterators to their underlying iterators is:

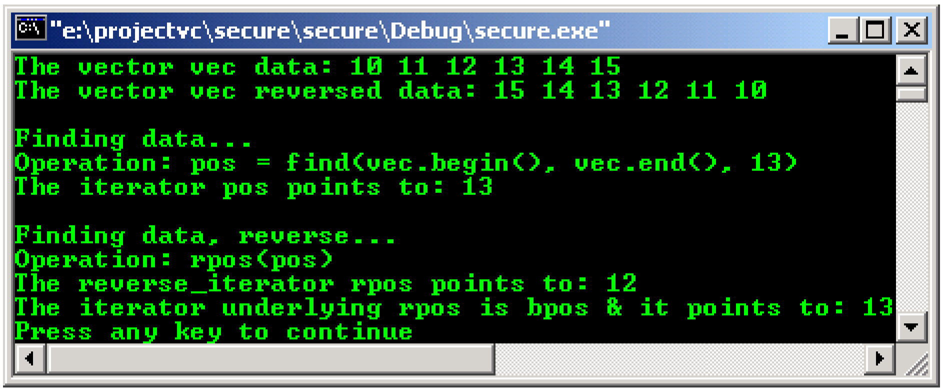
```
&*(reverse_iterator(i)) == &*(i – 1).
```

- In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence.
- So if an iterator addressed the element valued 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element valued 4 in the reversed sequence (8, 6, 4, 2).

```
//reverse_iterator, base()
#include <iterator>
#include <algorithm>
#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 vIter;
cout<<"The vector vec data: ";
for(vIter = vec.begin(); vIter != vec.end(); vIter++)
cout<<*vIter<<" ";
cout<<endl;

vector<int>::reverse_iterator rvIter;
cout<<"The vector vec reversed data: ";
for(rvIter = vec.rbegin(); rvIter != vec.rend(); rvIter++)
cout<<*rvIter<<" ";
cout<<endl;

cout<<"\nFinding data...";
cout<<"\nOperation: pos = find(vec.begin(), vec.end(), 13)\n";
vector <int>::iterator pos, bpos;
pos = find(vec.begin(), vec.end(), 13);
cout<<"The iterator pos points to: "<<*pos<<endl;

typedef reverse_iterator<vector<int>::iterator>::iterator_type it_vec_int_type;
cout<<"\nFinding data, reverse...\n";
cout<<"Operation: rpos(pos)\n";

reverse_iterator<it_vec_int_type> rpos(pos);
cout<<"The reverse_iterator rpos points to: "<<*rpos<<endl;
bpos = rpos.base();
cout<<"The iterator underlying rpos is bpos & it points to: "<<*bpos<<endl;
return 0;
}
```

Output:



"e:\projectvc\secure\secure\Debug\secure.exe"

```
The vector vec data: 10 11 12 13 14 15
The vector vec reversed data: 15 14 13 12 11 10

Finding data...
Operation: pos = find(vec.begin(), vec.end(), 13)
The iterator pos points to: 13

Finding data, reverse...
Operation: rpos(pos)
The reverse_iterator rpos points to: 12
The iterator underlying rpos is bpos & it points to: 13
Press any key to continue
```

Operators

Operator	Description
operator*	Returns the element that a reverse_iterator addresses.
operator+	Adds an offset to an iterator and returns the new reverse_iterator addressing the inserted element at the new offset position.
operator++	Increments the reverse_iterator to the next element.
operator+=	Adds a specified offset from a reverse_iterator.
operator-	Subtracts an offset from a reverse_iterator and returns a reverse_iterator addressing the element at the offset position.
Operator--	Decrements the reverse_iterator to the previous element.
operator-=	Subtracts a specified offset from a reverse_iterator.
operator->	Returns a pointer to the element addressed by the reverse_iterator.
operator[]	Returns a reference to an element offset from the element addressed by a reverse_iterator by a specified number of positions.

Table 32.28

32.2 Iterator Adapters

- We can write classes that have the interface of iterators but do something completely different. The C++ standard library provides several predefined special iterators, iterator adapters. They extend the functionalities of the iterators.
- The three iterator adapters are:

1. Insert iterators
2. Stream iterators
3. Reverse iterators

32.2.1 Insert Iterators

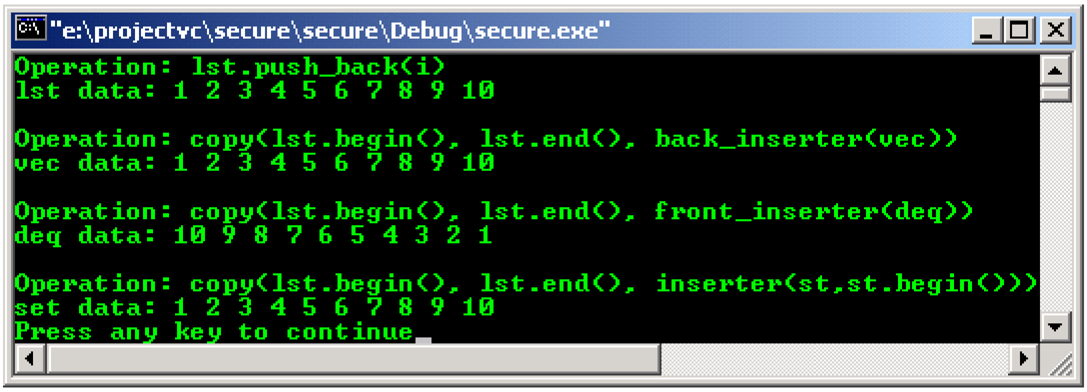
- Insert iterators, or inserters are used to let algorithms operate in the insert mode rather than in an overwrite mode.
- In particular, they solve the problem of algorithms that write to a destination that does not have enough storage; they let the destination grow accordingly.
- The following table lists the insert iterators and their functionality.

Insert iterator	Operation
`back_inserter(container)`	Appends in the same order by using `push_back()`
`front_inserter(container)`	Inserts at the front in reverse order by using `push_front()`
`inserter(container, pos)`	Inserts at `pos` (in the same order) by using `insert()`

32.29: Predefined insert iterators

```
//Inserter iterator
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
list<int> lst;
list <int>::iterator lstIter;
//insert elements from 1 to 10 into the lst list
for(int i=1; i<=10; ++i)
lst.push_back(i);

cout<<"Operation: lst.push_back(i)\n";
cout<<"lst data: ";
for(lstIter = lst.begin(); lstIter != lst.end(); lstIter++)
cout<<*lstIter<<' ';
cout<<endl;

//copy the elements of lst list into vec vector by appending them
vector<int> vec;
vector <int>::iterator Iter;
//from source to destination...
copy(lst.begin(), lst.end(), back_inserter(vec));

cout<<"\nOperation: copy(lst.begin(), lst.end(), back_inserter(vec))\n";
cout<<"vec data: ";
for(Iter = vec.begin(); Iter != vec.end(); Iter++)
cout<<*Iter<<" ";
cout<<endl;

//copy the elements of lst list into
//deq deque by inserting them at the front
//and reverses the order of the elements
deque<int> deq;
deque <int>::iterator deqIter;
copy(lst.begin(), lst.end(), front_inserter(deq));

cout<<"\nOperation: copy(lst.begin(), lst.end(), front_inserter(deq))\n";
cout<<"deq data: ";
for(deqIter = deq.begin(); deqIter != deq.end(); deqIter++)
cout<<*deqIter<<" ";
cout<<endl;

//copy elements of lst list into st set
//only inserter that works for associative collections
set<int> st;
set<int>::iterator stIter;
copy(lst.begin(), lst.end(), inserter(st, st.begin()));

cout<<"\nOperation: copy(lst.begin(), lst.end(), inserter(st, st.begin()))\n";
cout<<"set data: ";
for(stIter = st.begin(); stIter != st.end(); stIter++)
cout<<*stIter<<" ";
cout<<endl;
return 0;
}
```

Output:



```
Operation: lst.push_back(i)
lst data: 1 2 3 4 5 6 7 8 9 10

Operation: copy(lst.begin(), lst.end(), back_inserter(vec))
vec data: 1 2 3 4 5 6 7 8 9 10

Operation: copy(lst.begin(), lst.end(), front_inserter(deq))
deq data: 10 9 8 7 6 5 4 3 2 1

Operation: copy(lst.begin(), lst.end(), inserter(st,st.begin()))
set data: 1 2 3 4 5 6 7 8 9 10
Press any key to continue
```

- The program example uses all three predefined insert iterators as listed below:

Iterator	Description
Back inserters	Back inserters can be used only for containers that provide `push_back()` as a member function. In the C++ standard library, these containers are vector, deque, and list.
Front inserters	Front inserter reverses the order of the inserted elements. If you insert 1 at the front and then 2 at the front, the 1 is after the 2. Front inserters can be used only for containers that provide `push_front()` as a member function. In the C++ standard library, these containers are deque and list.
General inserters	A general inserter, also called simply an `inserter`, inserts elements directly in front of the position that is passed as the second argument of its initialization. It calls the `insert()` member function with the new value and the new position as arguments. Note that all predefined containers have such an `insert()` member function. This is the only predefined inserter for associative containers.

Table 32.30

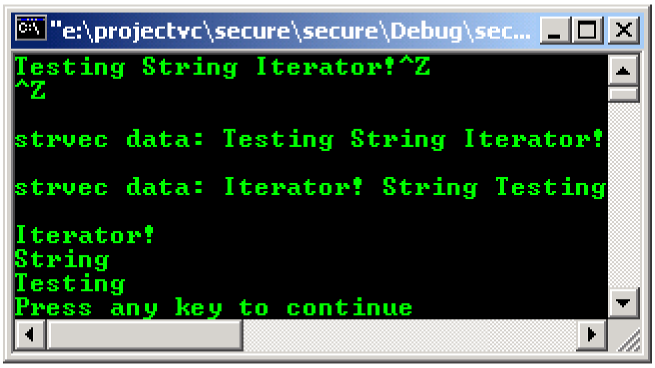
32.2.2 Stream Iterators

- Another very helpful kind of iterator adapter is a **stream iterator**. Stream iterators are iterators that read from and write to a stream.
- Thus, they provide an abstraction that lets the input from the keyboard behave as a collection, from which you can read. Similarly you can redirect the output of an algorithm directly into a file or onto the screen.
- Consider the following example. It is a typical example of the power of the whole STL. Compared with ordinary C or C++, it does a lot of complex processing by using only a few statements. For example study the following example.

```
//stream iterator
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
```

```
int main()
{
vector<string> strvec;
vector <string>::iterator Iter;
//read from the standard input until EOF/error
//the EOF is platform dependent...
//then copy (inserting) to strvec vector...
//copy from begin to end of source, to destination

copy(istream_iterator<string>(cin), istream_iterator<string>(), back_inserter(strvec));

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

//do some sorting
sort(strvec.begin(), strvec.end());

cout<<"\nstrvec data: ";
for(Iter = strvec.begin(); Iter != strvec.end(); Iter++)
cout<<*Iter<<" ";
cout<<"\n\n";

//print all elements without duplicates to standard output
unique_copy(strvec.begin(), strvec.end(), ostream_iterator<string> (cout, "\n"));
return 0;
}
```

Output:



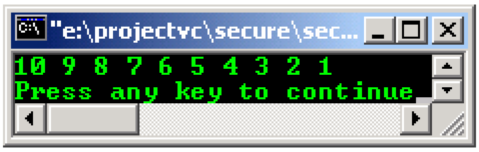
```
Testing String Iterator!^Z
^Z

strvec data: Testing String Iterator!

strvec data: Iterator! String Testing

Iterator!
String
Testing
Press any key to continue
```

32.2.3 Reverse Iterators

- The third kind of predefined iterator adapters are reverse iterators.
- Reverse iterators operate in reverse. They switch the call of an increment operator internally into a call of the decrement operator, and vice versa.
- All containers can create reverse iterators via their member functions `rbegin()` and `rend()`.

```
//reverse iterator using
//rbegin() and rend()
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
vector<int> vec;
//insert elements from 1 to 10
for(int i=1; i<=10; ++i)
vec.push_back(i);

//print all element in reverse order
copy(vec.rbegin(), vec.rend(), ostream_iterator<int> (cout," "));
cout<<endl;
return 0;
}
```

Output:



```
10 9 8 7 6 5 4 3 2 1
Press any key to continue
```

- Program example compiled using g++.

```
//******ostreamiterator.cpp********
//ostream_iterator, ostream_iterator
#include <iterator>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
//ostream_iterator for stream cout
ostream_iterator<int> intOut(cout, "\n");
*intOut = 12;
intOut++;
*intOut = 33;
intOut++;

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

cout<<"Operation: with and without delimiter...\n";
//Write elements to standard output stream
cout<<"Elements output without delimiter: ";
copy(vec.begin(), vec.end(), ostream_iterator<int> (cout));
cout<<endl;

//Write elements with delimiter " " to output stream
cout<<"Elements output with delimiter: ";
copy(vec.begin(), vec.end(), ostream_iterator<int> (cout, " "));
cout<<endl;
return 0;
}
```

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

```
12
33
Operation: with and without delimiter...
Elements output without delimiter: 101112131415
Elements output with delimiter: 10 11 12 13 14 15
```

```
//*****insertiter.cpp******
//Inserter iterator
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
list<int> lst;
list <int>::iterator lstIter;
//insert elements from 1 to 10 into the lst list
for(int i=1; i<=10; ++i)
lst.push_back(i);

cout<<"Operation: lst.push_back(i)\n";
cout<<"lst data: ";
for(lstIter = lst.begin(); lstIter != lst.end(); lstIter++)
cout<<*lstIter<<' ';
cout<<endl;

//copy the elements of lst list into vec vector by appending them
vector<int> vec;
vector <int>::iterator Iter;
//from source to destination...
copy(lst.begin(), lst.end(), back_inserter(vec));

cout<<"\nOperation: copy(lst.begin(), lst.end(), back_inserter(vec))\n";
cout<<"vec data: ";
for(Iter = vec.begin(); Iter != vec.end(); Iter++)
cout<<*Iter<<" ";
cout<<endl;

//copy the elements of lst list into
//deq deque by inserting them at the front
//and reverses the order of the elements
deque<int> deq;
deque <int>::iterator deqIter;
copy(lst.begin(), lst.end(), front_inserter(deq));

cout<<"\nOperation: copy(lst.begin(), lst.end(), front_inserter(deq))\n";
cout<<"deq data: ";
for(deqIter = deq.begin(); deqIter != deq.end(); deqIter++)
cout<<*deqIter<<" ";
cout<<endl;

//copy elements of lst list into st set
//only inserter that works for associative collections
set<int> st;
set<int>::iterator stIter;
copy(lst.begin(), lst.end(), inserter(st, st.begin()));

cout<<"\nOperation: copy(lst.begin(), lst.end(), inserter(st, st.begin()))\n";
cout<<"set data: ";
for(stIter = st.begin(); stIter != st.end(); stIter++)
cout<<*stIter<<" ";
cout<<endl;
return 0;
}
```

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

```
Operation: lst.push_back(i)
lst data: 1 2 3 4 5 6 7 8 9 10

Operation: copy(lst.begin(), lst.end(), back_inserter(vec))
vec data: 1 2 3 4 5 6 7 8 9 10

Operation: copy(lst.begin(), lst.end(), front_inserter(deq))
deq data: 10 9 8 7 6 5 4 3 2 1

Operation: copy(lst.begin(), lst.end(), inserter(st, st.begin()))
set data: 1 2 3 4 5 6 7 8 9 10
```

---End of Iterator---

---www.tenouk.com---

Further reading and digging:

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