```
1  // Demonstrate the front_inserter, back_inserter, ostream_iterator,
2  // and functors.
3  // George F. Riley, Georgia Tech, Fall 2009
4
5  #include <iostream>
6  #include <algorithm>
7  #include <vector>
8  #include <set>
9  #include <deque>
10 #include <iterator>
11
12 using namespace std;
13
14 // Some of the STL algorithms, such as "copy" take three iterators
15 // as arguments; the first two define the begin() and end() of
16 // a container, and the third defines an iterator to use to
17 // insert elements into a new container.  A possible implementation
18 // is below:
19
20 template <typename InputIterator, typename OutputIterator >
21 void Copy(InputIterator b, InputIterator e, OutputIterator d)
22 { // b is begin of input sequence, e is end of input
23   // d is destination iterator
24   cout << "Hello from Copy" << endl;
25   while(b != e)
26     {
27       *d++ = *b++;
28     }
29 }
30
31 // Generic subroutine to print a container
32 template <class ForwardIterator>
33 void Print(ForwardIterator b, ForwardIterator e, bool addEndl = true)
34 {
35   while(b != e)
36     {
37       cout << (*b++);
38       if (addEndl) cout << endl;
39       else         cout << " ";
40     }
41 }
42
43 // Now we look at some special objects called "functors".
44 // A functor is simply a class defines the "()" operator.
45 template <typename T> class Greater
46 {
47 public:
48   bool operator()(const T& lhs, const T& rhs) const { return lhs > rhs; }
49 };
50
51 int main()
52 {
53   // Create a short vector
54   vector<int> v;
55   v.push_back(1);
56   v.push_back(2);
```

Program special-iterators.cc

```
57    v.push_back(3);
58    // Create an empty vector
59    vector<int> v1;
60    // Now we want to copy the contents of the first vector to the second
61    // using Copy.  The problem is, what do we use for the third argument
62    // to Copy.  One guess would be "v1.end()", indicating  that we want
63    // to insert at the end of the new vector.
64
65    // Copy(v.begin(), v.end(), v1.end()); // Like this
66
67    // While this is exactly what we want, the code fails miserably.
68    // Recall that "end()" is an
69    // iterator that points "one beyond the end" of the vector, and in fact
70    // does not point to a valid element.  Thus in Copy when we say "*d++",
71    // we are dereferencing an invalid iterator, with unpredictable results.
72    //
73    // We could try passing v1.begin() as the third argument.
74
75    // Copy(v.begin(), v.end(); v1.begin()); // Like this
76
77    // If v1 already had v.size() elements, this would work.  But if
78    // v1 has fewer elements (which it does in this example), the
79    // code again crashes, since Copy advances the output iterator
80    // v.size() times which extends beyond the end of v1.
81
82    // Luckily, the designers of the STL anticipated our need, and designed
83    // two special iterators that handle this situation.  They are the
84    // "back_insert_iterator" and the "front_insert_iterator"
85    back_insert_iterator<vector<int> > bi(v1);
86    // Note the extra space between the two > characters in the declaration
87    // above.  This is necessary, otherwise the compiler would parse
88    // a right shift operator ">>" which  not valid in this context.
89    // The variable "bi" is a back insert iterator, which has semantics
90    // exactly like normal iterators, except that if dereferenced at
91    // the end of a container, it adds the element to the cotainer.
92    Copy(v.begin(),  v.end(), bi);
93    cout << "v is  "; Print( v.begin(),  v.end(), false); cout << endl;
94    cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
95    // We can also dereference and increment the back_insert_iterator
96    cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
97    *bi++ = 4;
98    cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
99    *bi++ = 5;
100   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
101   // And we can dereference without incrementing (same results)
102   *bi++;
103   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size (no assignment)" <<
104   *bi = 6;
105   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
106   *bi = 7;
107   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
108   *bi = 8;
109   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << e
110   // But we can't decrement it (doesn't make sense and doesn't compile)
111   //*bi-- = 6;
112   cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
```

Program special-iterators.cc (continued)

```
113
114     // There is also a special shortcut for creating the back_insert_iterator,
115     // called "back_inserter".  back_inserter is simply a global function
116     // that has a single argument of any container, and returns a
117     // back_insert_iterator for that container.  So instead of defining
118     // "bi" separately, we could simply say:
119     Copy(v.begin(), v.end(), back_inserter(v1));
120     cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
121
122     // Similarly, we can define a "front_insert_iterator" that will
123     // add elements to the front (beginning) of a container when
124     // dereferenced.  However, we have to be careful, since the
125     // container being inserted must have a "push_front" operation
126     // defined, which vectors do not.  If we tried to define and
127     // use a front_insert_iterator on a vector container, the compiler
128     // would complain.  We can however use a front insert iterator
129     // on the "deque" container, which supports both push_back
130     // and "pop_front".
131     deque<int> d1;
132
133     front_insert_iterator<deque<int> > fi(d1);
134     Copy(v.begin(), v.end(), fi);
135     cout << "v is  "; Print( v.begin(),  v.end(), false); cout << endl;
136     cout << "d1 is "; Print(d1.begin(), d1.end(), false); cout << endl;
137     // Similarly, we can use the "front_inserter" shortcut:
138     Copy(v.begin(), v.end(), front_inserter(d1));
139     cout << "d1 is "; Print(d1.begin(), d1.end(), false); cout << endl;
140
141     // It is often the case that we use iterators for a sequence to
142     // call an ostream function (for example cout <<) for each element.
143     // There is a nice shortcut to do this called "ostream_iterator"
144     ostream_iterator<int> osi(cout, ", ");
145     // By passing this as the destination iterator for Copy, we cause
146     // the cout << operator to be called for each element.
147     Copy(v1.begin(), v1.end(), osi); cout << endl;
148     Copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, "\n"));
149
150     // Now demonstrate the use of functors
151     Greater<int> greater;// Class Greater has one member function, the operator()
152     if (greater(1,2)) cout << "Huh? 1 greater than 2???" << endl;
153     else              cout << "1 not greater than 2" << endl;
154     // The above is not very intersting, as it's not clear what value
155     // the functors are.  Recall the STL "set" container that stores
156     // objects in sorted order.  The default comparator for set containers
157     // is the "operator <" function.
158     set<int> s;
159     s.insert(100);
160     s.insert(10);
161     s.insert(500);
162     // This stores in normal ascending order
163     Copy(s.begin(), s.end(), ostream_iterator<int>(cout, " ")); cout << endl;
164     // But we can create a set specifying a non-default comparator,
165     // by passing the Greater functor:
166     set<int, Greater<int> > s1;
167     s1.insert(100);
168     s1.insert(10);
```

Program special-iterators.cc (continued)

```
169     s1.insert(500);
170     Copy(s1.begin(), s1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
171   }
```

Program special-iterators.cc (continued)